# REAL-TIME ASP.NET CORE 3 APPS WITH SIGNALR

SUCCINCTLY

BY DIRK STRAUSS

SUCCINCTLY EBOOK SERIES

Syncfusion

# Real-Time ASP.NET Core 3 Apps with SignalR Succinctly

**By**
**Dirk Strauss**

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA


**Technical Reviewer:** James McCaffrey
**Copy Editor:** Courtney Wright
**Acquisitions Coordinator:** Tres Watkins, vice president of content, Syncfusion, Inc.
**Proofreader:** Jacqueline Bieringer, content producer, Syncfusion, Inc.

# Table of Contents

**The Story behind the *Succinctly* Series of Books** ..... **5**

**About the Author** ..... **7**

**Chapter 1 What is SignalR?** ..... **8**

Hubs ..... 10

**Chapter 2 Prerequisites** ..... **17**

Visual Studio ..... 17

Visual Studio Code ..... 19

Visual Studio for Mac ..... 21

**Chapter 3 Project Creation and Setup** ..... **22**

**Chapter 4 Add the Required SignalR Client Library** ..... **26**

**Chapter 5 Creating the Hub and Configuring SignalR** ..... **30**

Configuring SignalR ..... 32

The Startup class ..... 34

**Chapter 6 Creating the Client Application** ..... **36**

**Chapter 7 Running the Application** ..... **41**

**Chapter 8 The Problem with SendAsync** ..... **46**

**Chapter 9 Creating Real-Time Charts** ..... **49**

**Chapter 10 Creating the Chart Hub** ..... **51**

**Chapter 11 Creating the Chart Client Application** ..... **56**

**Chapter 12 The JavaScript Code Explained** ..... **64**

**Chapter 13 Running the Real-Time Chart Application** ..... **67**

**Chapter 14 Having Fun with Exchange Rates** ..... **69**

**Chapter 15 Using Chrome Developer Tools** ..... **75**

**Conclusion** ..... **81**

# The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

## Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

## Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

## The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

## The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

## Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

## Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

## Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

# About the Author

Dirk Strauss is a software developer from South Africa with over 13 years of programming experience. He has extensive experience in SYSPRO customization, with a main focus on C# and web development. He studied at Nelson Mandela University, where he wrote software part-time to gain a better understanding of the technology. He remains passionate about writing code and sharing what he learns with others.

# Chapter 1  What is SignalR?

The ASP.NET Core SignalR library is open source and allows developers to simplify adding real-time web functionality to applications. This means that the server-side code can instantly push content to connected clients.

Examples of applications that would be a good fit for SignalR include:

- Chat applications such as Facebook Messenger.
- Social network applications, such as Facebook, that display new notifications as they happen.
- Live dashboards, such as KPI dashboards, that instantly display updated sales information.
- Collaboration software that allows teams to meet and share ideas.

In essence, the SignalR API allows server-side code to call JavaScript functions on connected clients via remote procedure calls (RPCs).

SignalR for ASP.NET Core automatically takes care of connection management. If you think of a chat room example, SignalR can send messages to all connected clients at the same time. It can also send messages to a specific client or group of clients.

Later on, you will see how this is done when we look at the **`IHubCallerClients`** interface properties and methods.

SignalR for ASP.NET Core can automatically scale to allow it to handle increasing traffic.

***Note: You can find the SignalR repository on GitHub.***

To handle real-time communication, SignalR supports:

- WebSockets
- Server-Sent Events
- Long Polling

These are called transports. As shown in Figure 1, SignalR lies on top of the lower-level transports.


*Figure 1: SignalR Layer and Transports Layer*

Transports allow developers to focus on the business process instead of how the messages are sent (or deciding what transport to use). SignalR can dynamically evaluate the supported transports. In other words, SignalR will automatically choose which transport method is best suited, based on the capabilities of the client and server.

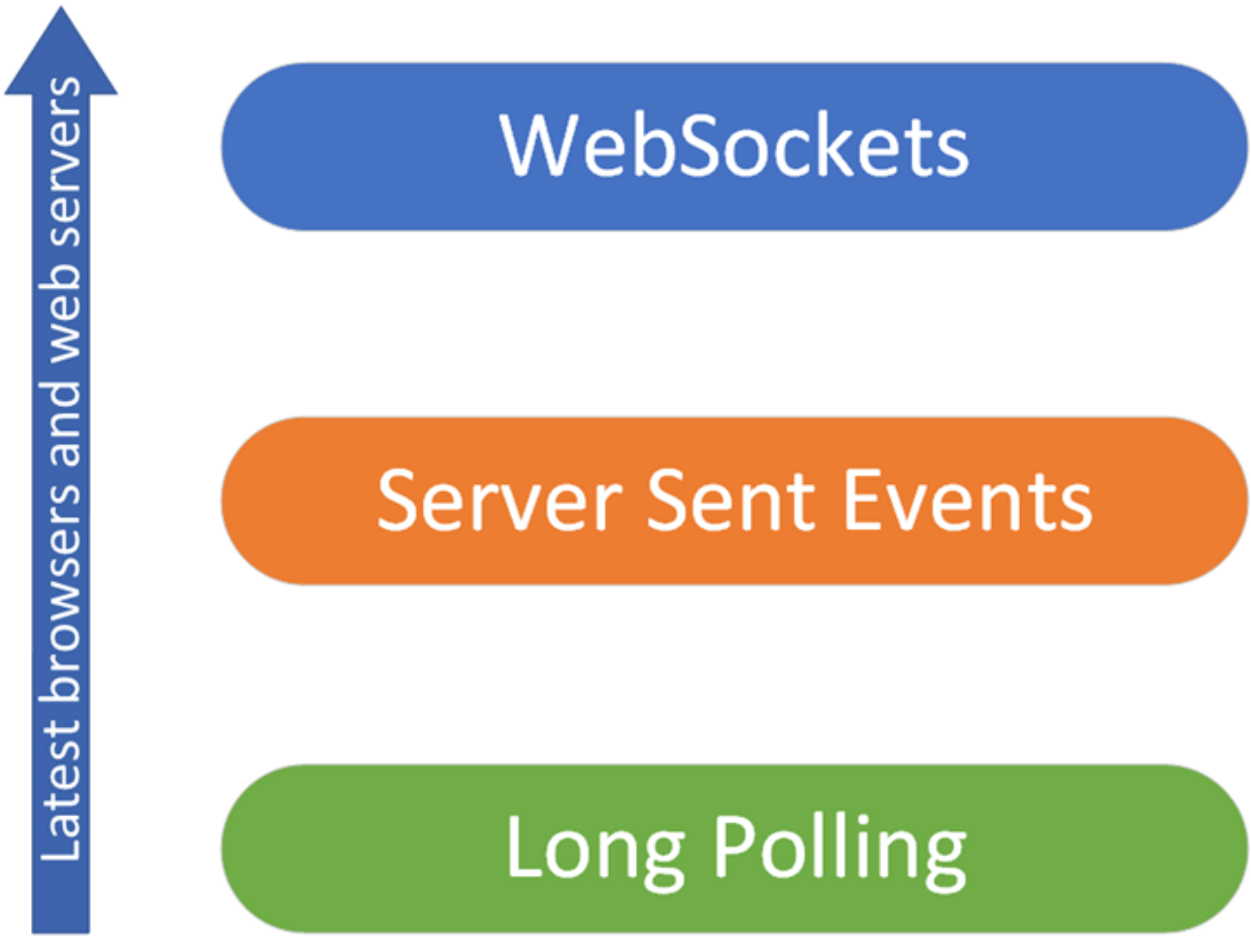


*Figure 2: SignalR Transports*

This can be illustrated as seen in Figure 2. If WebSockets (the most efficient of the transports) are not supported by the server or browser, SignalR will fall back on Server-Sent Events. If Server-Sent Events are not supported, SignalR will fall back on Long Polling.

> ***Note: According to the Internet Engineering Task Force (IETF) Internet Standards track document, the WebSocket protocol enables two-way, real-time communication between clients and servers in web-based applications.***

When SignalR establishes a connection, it will start sending keep-alive messages to verify that the connection is still alive. If the connection is not alive, SignalR will throw an exception.

Referring back to Figure 1, you will remember that SignalR lies on top of the transports. This allows developers to work in a consistent manner, regardless of which transport SignalR is using.

## Hubs

Hubs are used by SignalR to communicate between servers and clients. Later in this book, you will create a **`ChatHub`** class that inherits from the SignalR **`Hub`** class. The hub allows a client and server to call methods on each other.

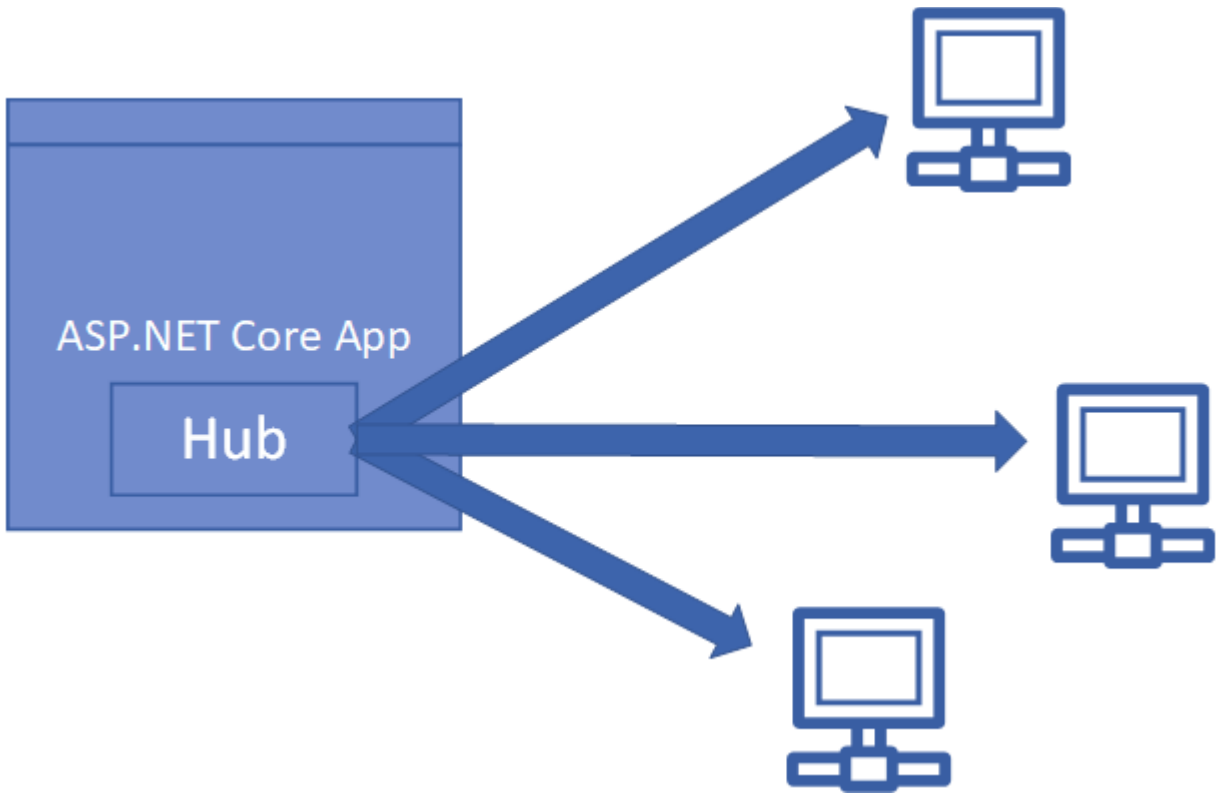


*Figure 3: Hub Messaging All Clients*

Inside the methods of this **Hub** class, you can call a method on all the connected clients, as seen in Figure 3.

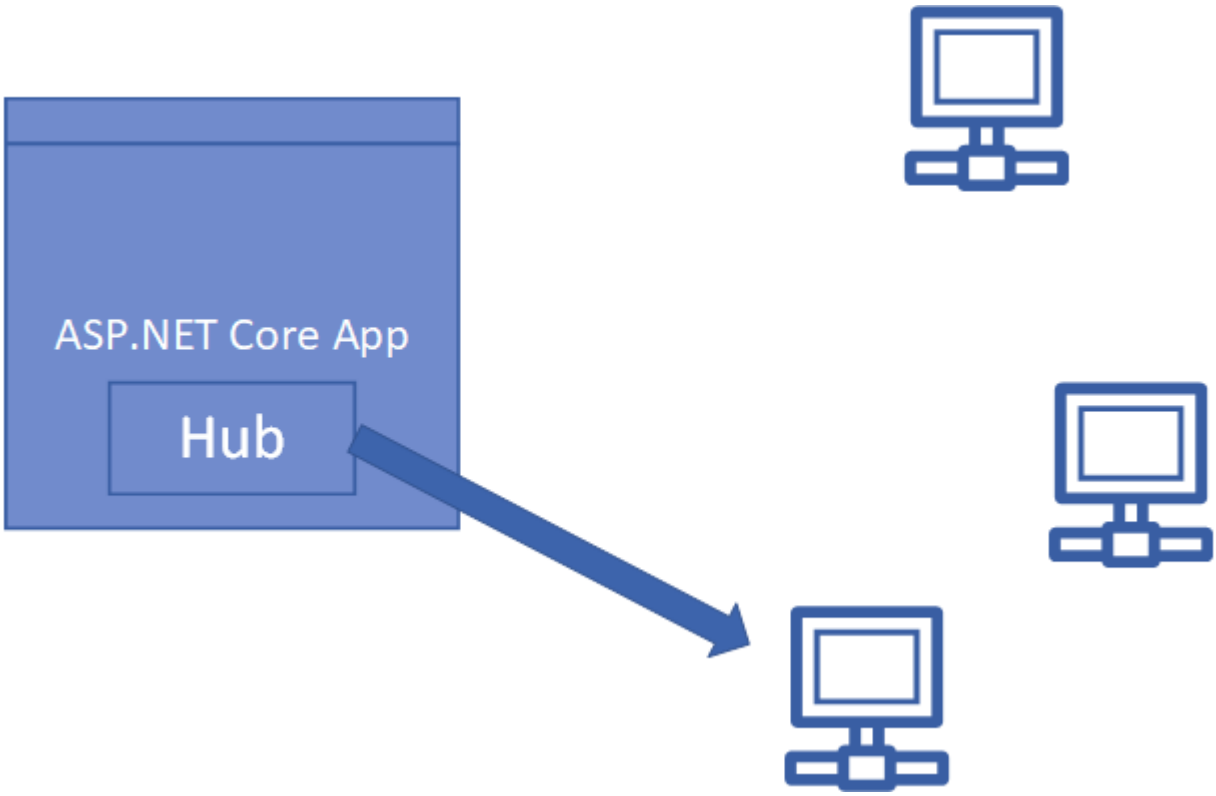


*Figure 4: Hub Messaging a Single Client*

The hub can also call a method on a specific client, as illustrated in Figure 4.

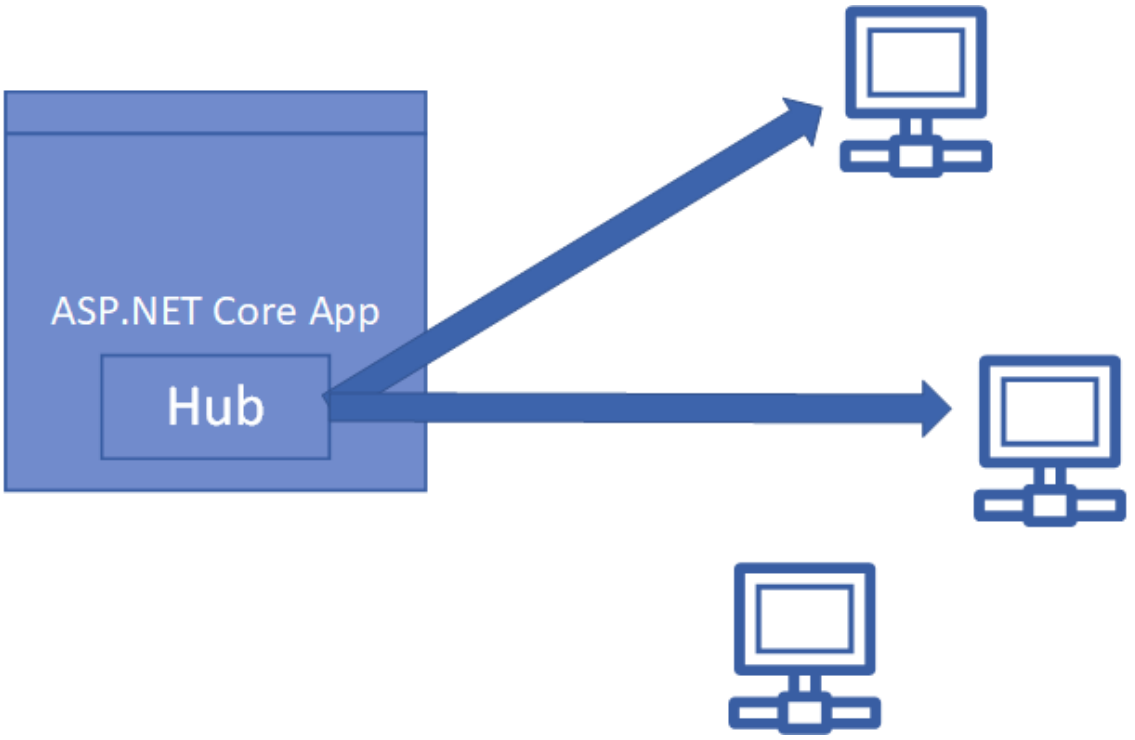


*Figure 5: Hub Messaging a Group of Clients*

The hub can also call a method on a group of clients, as illustrated in Figure 5.

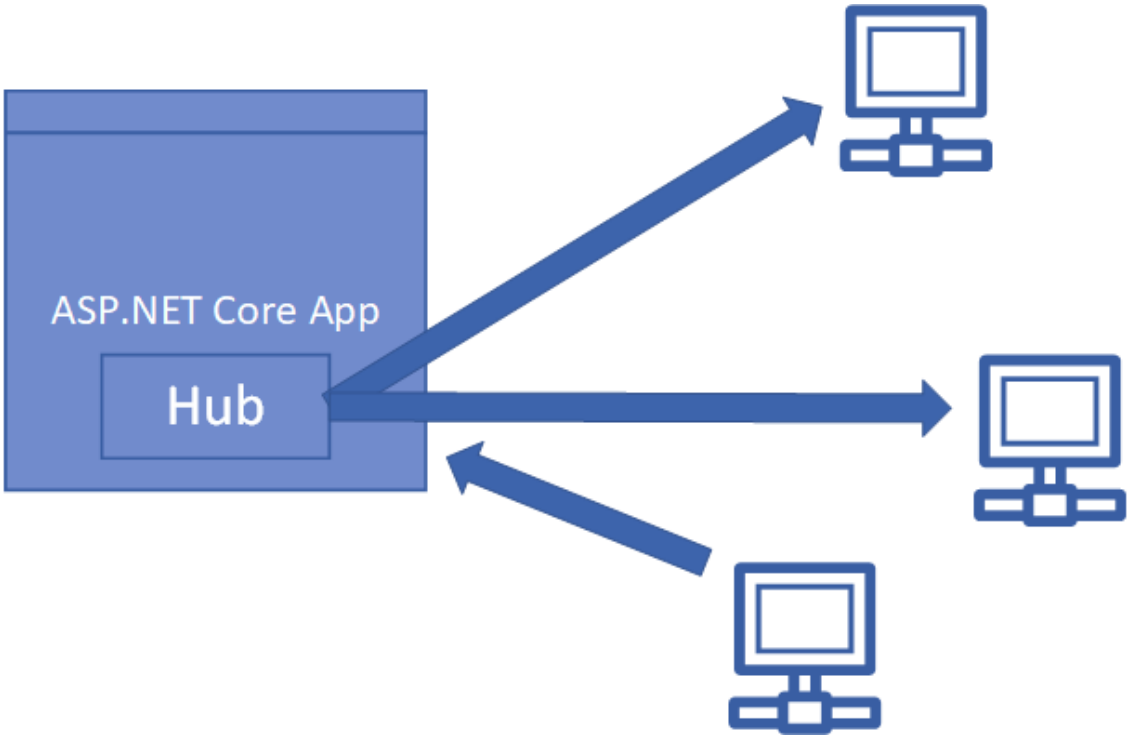


*Figure 6: Client Calling Hub Method*

Additionally, the client can call a method on the hub, as illustrated in Figure 6.

You will see the code in action later in this book, but I want to briefly discuss the configuration of SignalR hubs.

> ***Note: In ASP.NET Core, the `Startup` class configures services, as well as the application's request pipeline. To configure these services, you must register them in the `ConfigureServices` method. Registered services are then used throughout the app via dependency injection.***

We configure the SignalR middleware by registering the following service in the **`ConfigureServices`** method of the **`Startup`** class.

*Code Listing 1*

```
services.AddSignalR();
```

We then need to set up the SignalR routes by calling **`endpoints.MapHub`** in the **`Configure`** method of the **`Startup`** class.

*Code Listing 2*

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
    endpoints.MapHub<ChatHub>("/chatHub");
});
```

The **`Hub`** class also has a **`Context`** property containing the following properties, which provide information about the connection:

- **`ConnectionAborted`**
- **`ConnectionId`**
- **`Features`**
- **`Items`**
- **`User`**
- **`UserIdentifier`**

The **`Context`** also contains an **`Abort`** method, as well as a **`GetHttpContext`** extension method. The **`HubCallerContext`** abstract class metadata is shown in Code Listing 3.

*Code Listing 3*

```
namespace Microsoft.AspNetCore.SignalR
{
    //
    // Summary:
    //     A context abstraction for accessing information about the hub
    //     caller connection.
```

```
    public abstract class HubCallerContext
    {
        protected HubCallerContext();

        //
        // Summary:
        //     Gets a System.Threading.CancellationToken that notifies
when
        //     the connection is
        //     aborted.
        public abstract CancellationToken ConnectionAborted { get; }
        //
        // Summary:
        //     Gets the connection ID.
        public abstract string ConnectionId { get; }
        //
        // Summary:
        //     Gets the collection of HTTP features available on the
connection.
        public abstract IFeatureCollection Features { get; }
        //
        // Summary:
        //     Gets a key/value collection that can be used to share data
within
        //     the scope of
        //     this connection.
        public abstract IDictionary<object, object> Items { get; }
        //
        // Summary:
        //     Gets the user.
        public abstract ClaimsPrincipal User { get; }
        //
        // Summary:
        //     Gets the user identifier.
        public abstract string UserIdentifier { get; }

        //
        // Summary:
        //     Aborts the connection.
        public abstract void Abort();
    }
}
```

The **`GetHttpContext`** method signature metadata is shown in Code Listing 4.

*Code Listing 4*

```
namespace Microsoft.AspNetCore.SignalR
```

```
{
    //
    // Summary:
    //     Extension methods for accessing
Microsoft.AspNetCore.Http.HttpContext
    //     from a hub context.
    public static class GetHttpContextExtensions
    {
        //
        // Summary:
        //     Gets Microsoft.AspNetCore.Http.HttpContext from the
specified
        //     connection, or
        //     null if the connection is not associated with an HTTP
request.
        //
        // Parameters:
        //   connection:
        //     The connection.
        //
        // Returns:
        //     The Microsoft.AspNetCore.Http.HttpContext for the
connection,
        //     or null if the
        //     connection is not associated with an HTTP request.
        public static HttpContext GetHttpContext(this HubCallerContext
connection);
        //
        // Summary:
        //     Gets Microsoft.AspNetCore.Http.HttpContext from the
specified
        //     connection, or
        //     null if the connection is not associated with an HTTP
request.
        //
        // Parameters:
        //   connection:
        //     The connection.
        //
        // Returns:
        //     The Microsoft.AspNetCore.Http.HttpContext for the
connection,
        //     or null if the
        //     connection is not associated with an HTTP request.
        public static HttpContext GetHttpContext(this
HubConnectionContext connection);
    }
}
```

The **`Hub`** class has a **`Clients`** property, but we will have a look at this in more detail when we create our real-time chat application.

One thing to note is that hubs are impermanent, which is to say that they only last for a short time. Because hubs are transient, you should not store state in a property on the hub class. This is because every hub method call is executed on a new instance of the hub.

When we create our **`ChatHub`** class later on, you will see that we use the syntactically required **`await`** keyword when calling **`Clients.All.SendAsync`**. This is because the **`SendAsync`** method could potentially fail if the hub method completes before the **`SendAsync`** finishes.

# Chapter 2 Prerequisites

To get started, you will need to ensure that you have a few prerequisites installed. Then we will see how to create a real-time ASP.NET Core 3.0 chat application.

## Visual Studio

If you want to use Visual Studio, ensure that you have Visual Studio 2019 with the ASP.NET and web development workload installed.

> ***Note: Workloads allow you to modify Visual Studio to include additional components for development.***

To get to your workloads, you need to use the Visual Studio Installer.

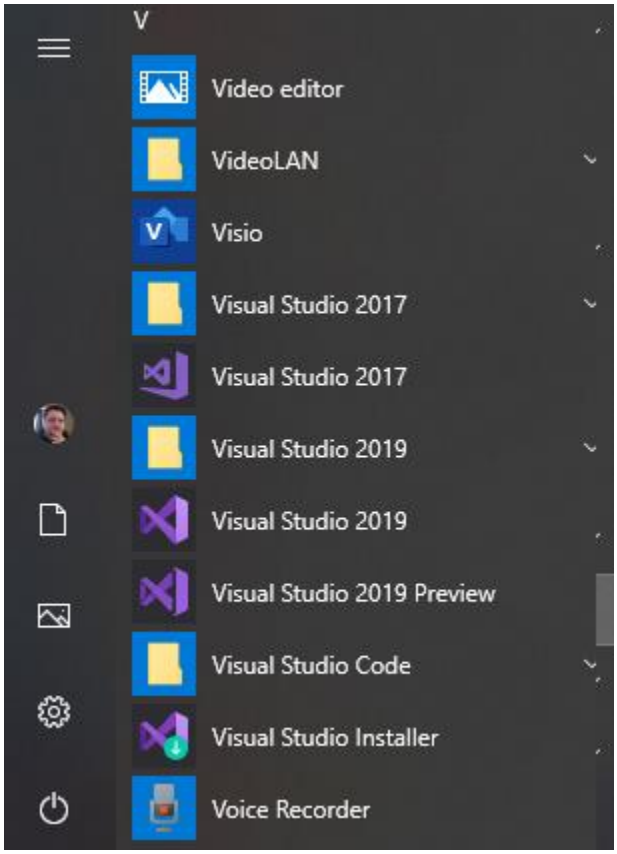


*Figure 7: Launch Visual Studio Installer*

Once the Visual Studio Installer is open, it will display all the installed versions of Visual Studio. Select the **Visual Studio 2019** version (I say this because you might have a preview version of Visual Studio 2019 installed) and click **Modify**.

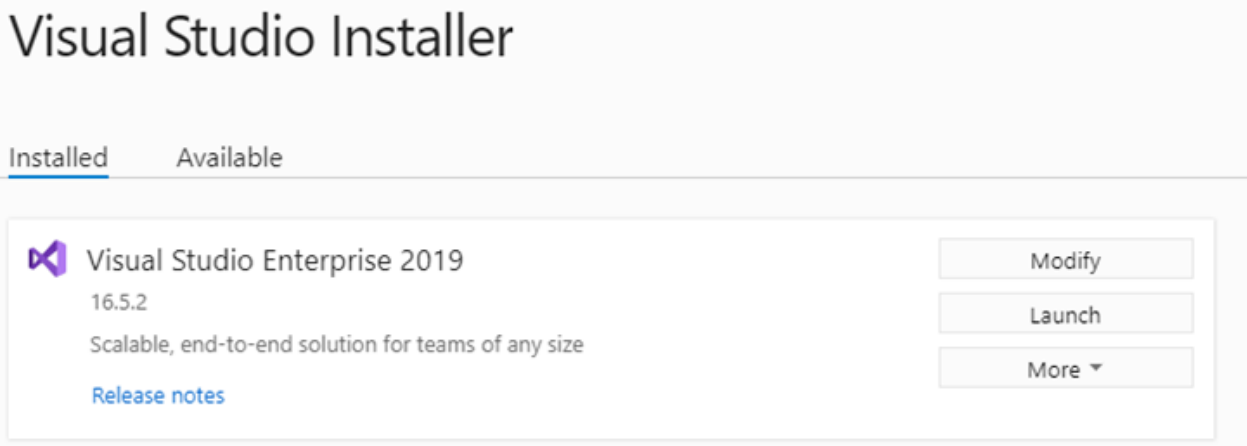


*Figure 8: Installed Apps*

This will display all the available workloads for you. You can view individual components, workloads, language packs, and installation components. Check the **ASP.NET and web development** workload check box and click **Modify**.

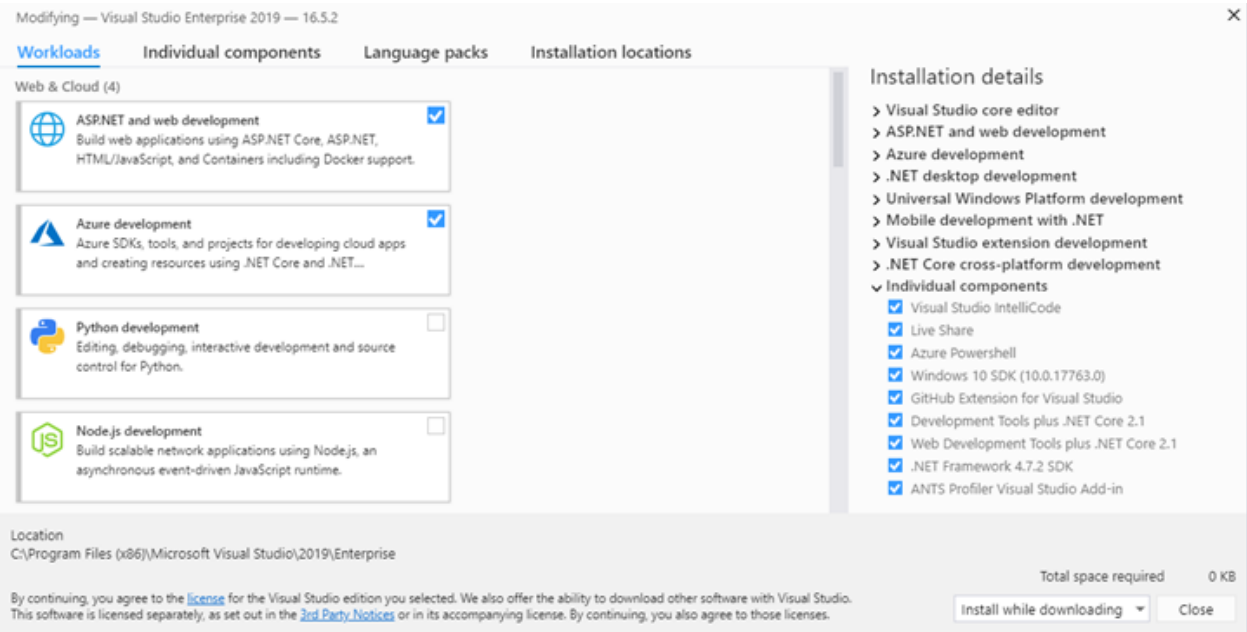


*Figure 9: Visual Studio 2019 Workloads*

In Figure 9, the screenshot does not display the Modify button because I already have the ASP.NET and web development workload installed. If you check a workload that isn't installed on your machine, the button at the bottom of the workloads screen will change, as shown in Figure 10.

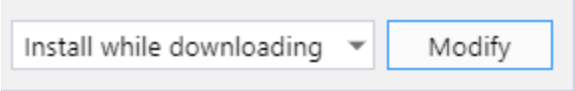


*Figure 10: Modify Button on Workloads Screen*

With the ASP.NET and web development workload installed, you need to make sure that you have the .NET Core 3.0 SDK or later installed. To do this, open your Command Prompt, and type **`dotnet --list-sdks`**, and select **Enter**.

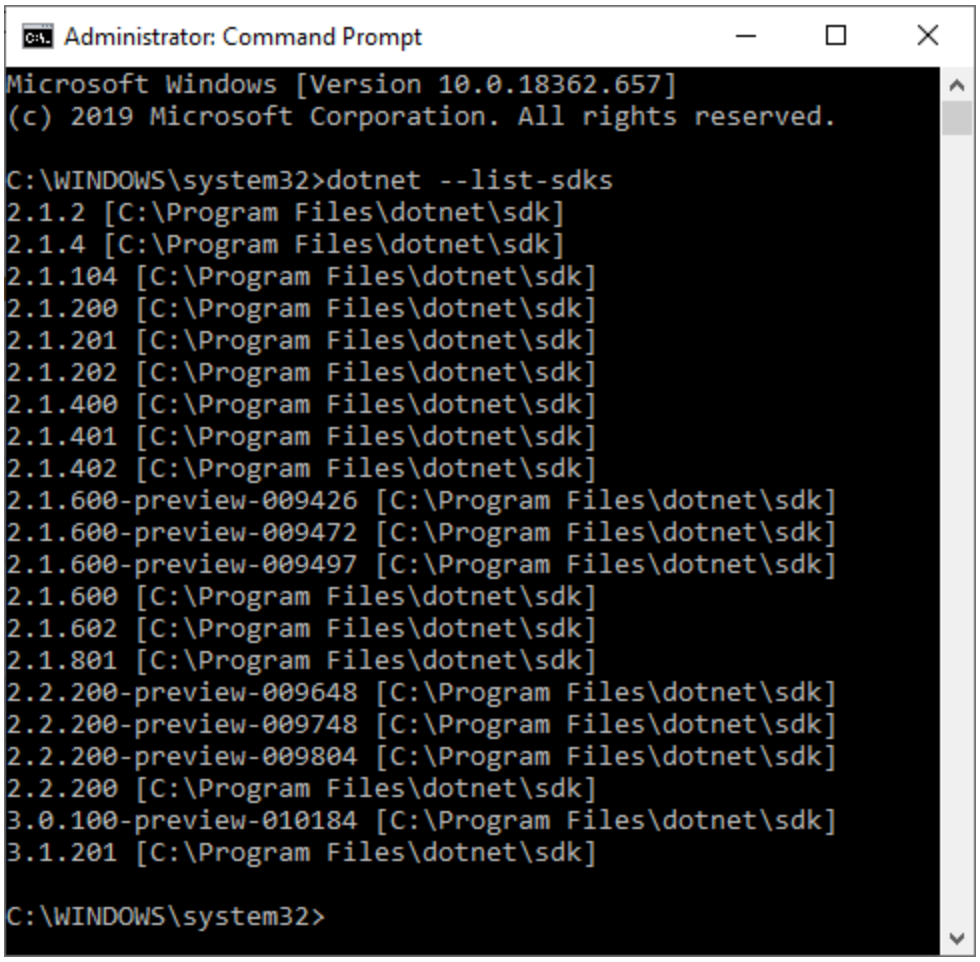


*Figure 11: Command Prompt Listing .NET Core SDKs*

This will list the .NET Core SDKs currently installed on your system. If you don't see the .NET Core 3.0 SDK or later installed, you need to head over to the download page for the current version of .NET Core (here at the time of writing).

## Visual Studio Code

If you want to use Visual Studio Code, head on over to the Visual Studio Code downloads page and grab a copy of the version for the system you are using (here at the time of writing).

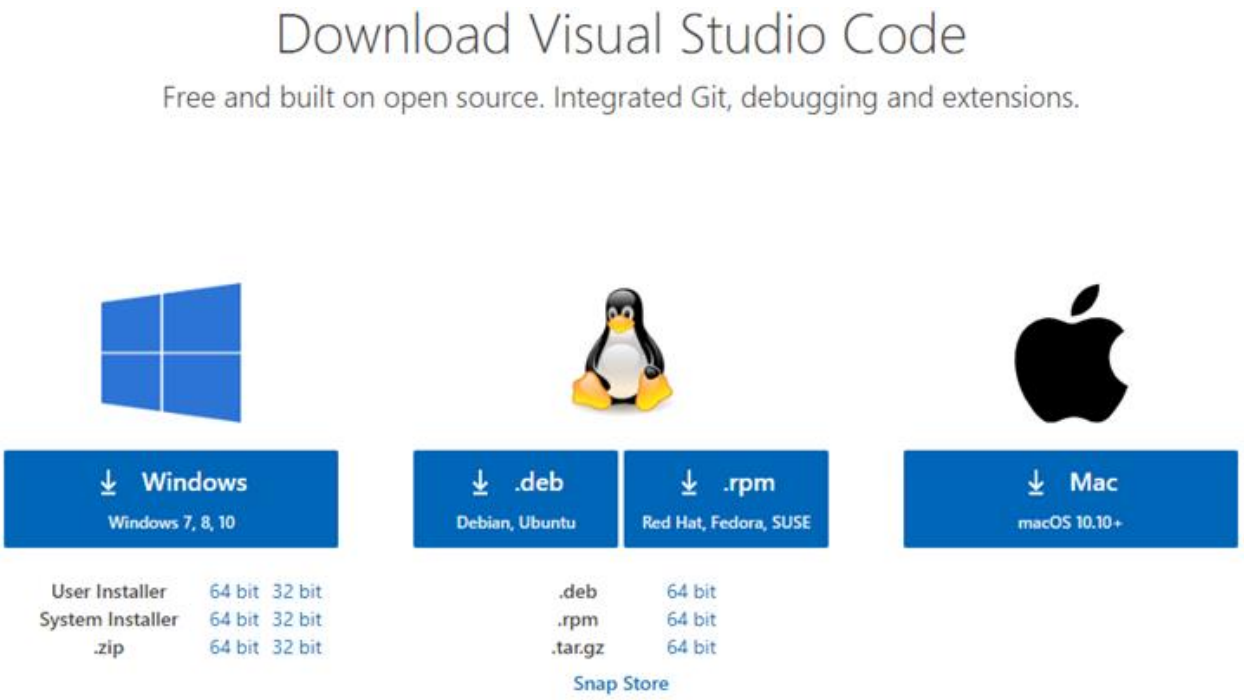


*Figure 12: Download Visual Studio Code*

Visual Studio Code users will also need to install the latest version of C# for Visual Studio Code. You can install it from the marketplace, or from within Visual Studio Code via the Extensions tab.

With Visual Studio Code open, press **Ctrl+Shift+X** and search for **C#**. You will see that the first result displayed is C# for Visual Studio Code by Microsoft (see Figure 13).

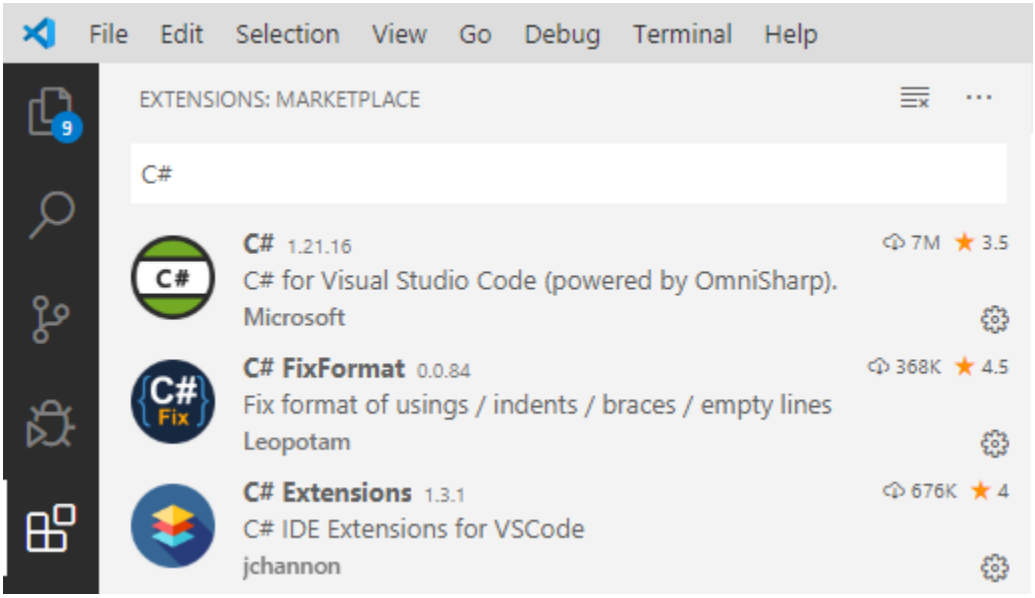


*Figure 13: Visual Studio Code Extensions*

Visual Studio Code users also need to install the .NET Core 3.0 SDK or later.

***Note: The previous section on Visual Studio explains how to download and install the .NET Core 3.0 SDK.***

Users on Mac can also use Visual Studio for Mac. Let's have a look at the prerequisites for that next.

## Visual Studio for Mac

If you want to run Visual Studio on your Mac to develop real-time ASP.NET Core applications using SignalR, you will need to download Visual Studio 2019 for Mac first.

You can download Visual Studio for Mac from this page, where you can also see a comparison between Visual Studio 2019 and Visual Studio 2019 for Mac.

Along with Visual Studio 2019 for Mac, you will need to install the .NET Core 3.0 SDK or later.

***Note: The previous section on Visual Studio explains how to download and install the .NET Core 3.0 SDK.***

Once you have all the prerequisites and components downloaded and installed, you are ready to create your web application.

# Chapter 3  Project Creation and Setup

In Visual Studio 2019, create a new project by going to **File** > **New Project**, or simply select the **Create a new project** option after launching Visual Studio.

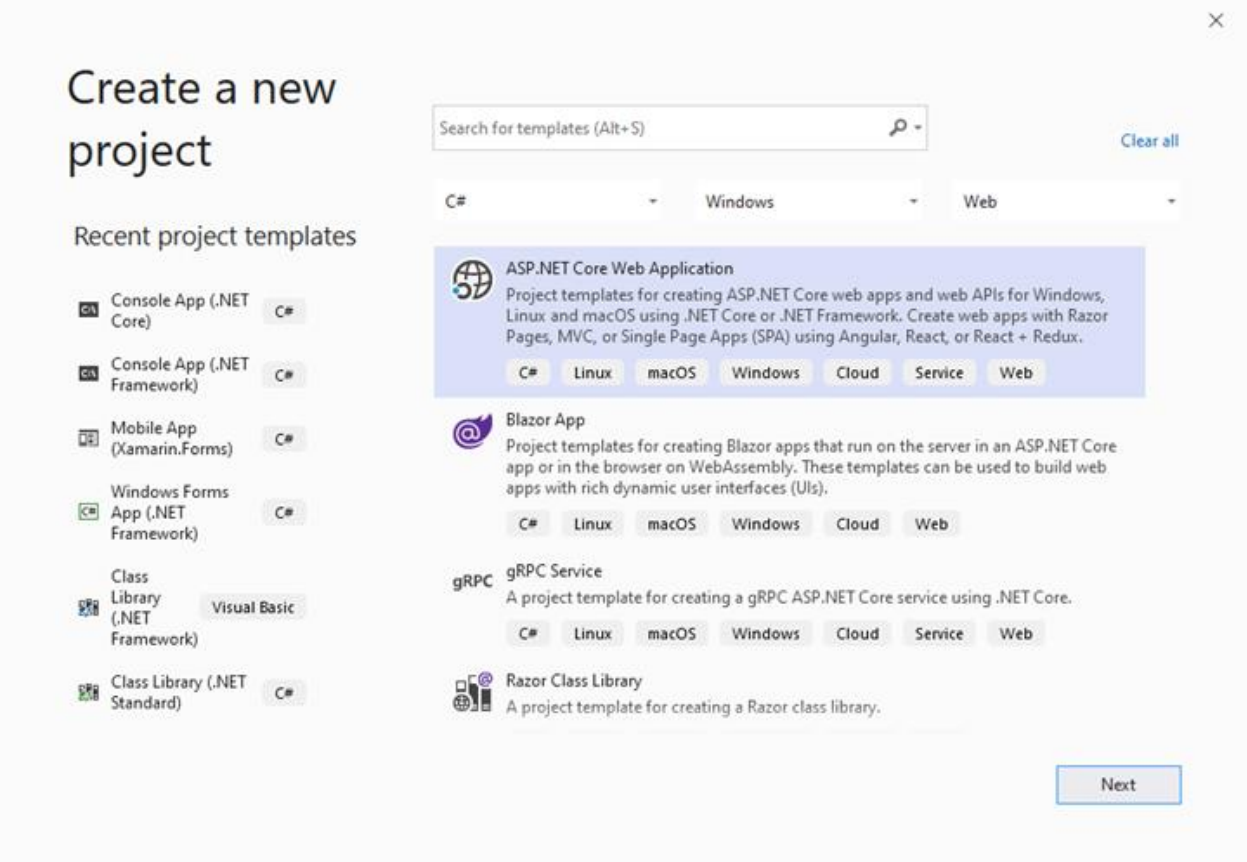


*Figure 14: Create a New Project*

The new project form will be displayed, and you will see that you can filter the project templates displayed by choosing C#, Windows, and Web from the drop-down filters.

Select the **ASP.NET Core Web Application** project template and click **Next**.

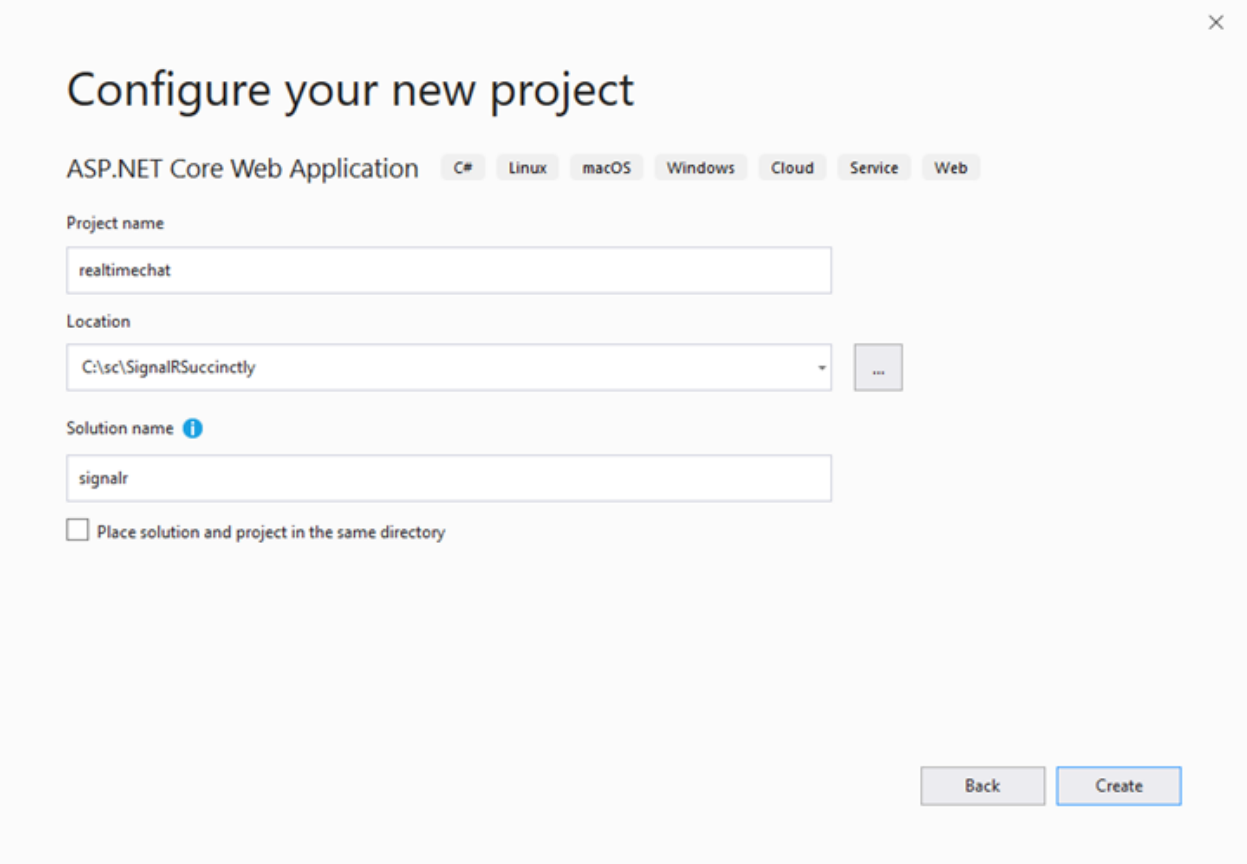


*Figure 15: Configuring Your Project*

The **Configure your new project** form will now be displayed, as seen in Figure 15. I called my project **realtimechat** and placed it into a solution called **signalr**. Specify the file location for your source code and click **Create**.

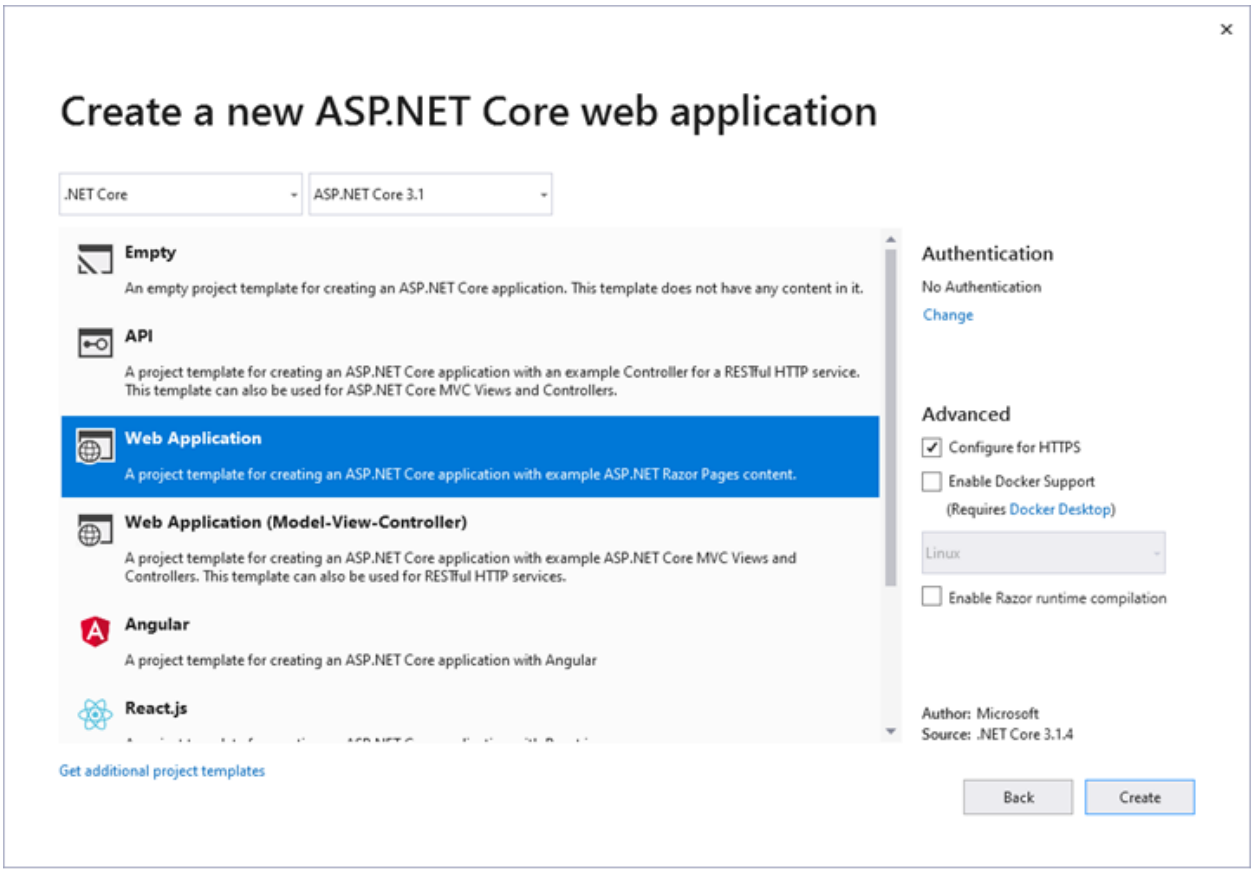


*Figure 16: Creating a .NET Core Web App*

You will then see the ASP.NET Core web application creation form. You want to select **.NET Core** and **ASP.NET Core 3.1** from the drop-down boxes at the top of the form. Next, you need to select the **Web Application** option from the list of project templates.

You can leave all the other options, such as Authentication, Docker Support, and Enable Razor runtime compilation, set to their default values. Click **Create** to create your ASP.NET Core Web Application project.

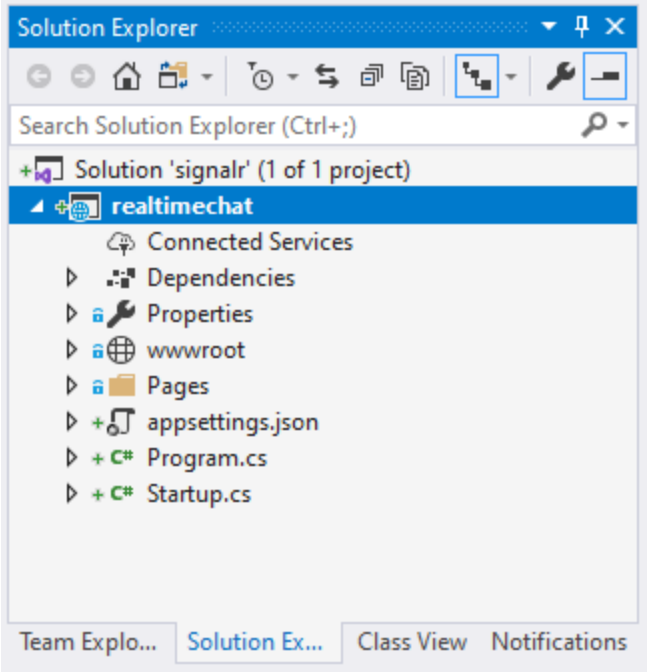


*Figure 17: The Created Solution and Project*

Once your Visual Studio project is created, your Solution Explorer window will look as shown in Figure 17. The next step we need to take is to add the SignalR client library to the project. Let's do that next.

# Chapter 4 Add the Required SignalR Client Library

We now need to add the JavaScript client library to the project. This isn't automatically included in the project.

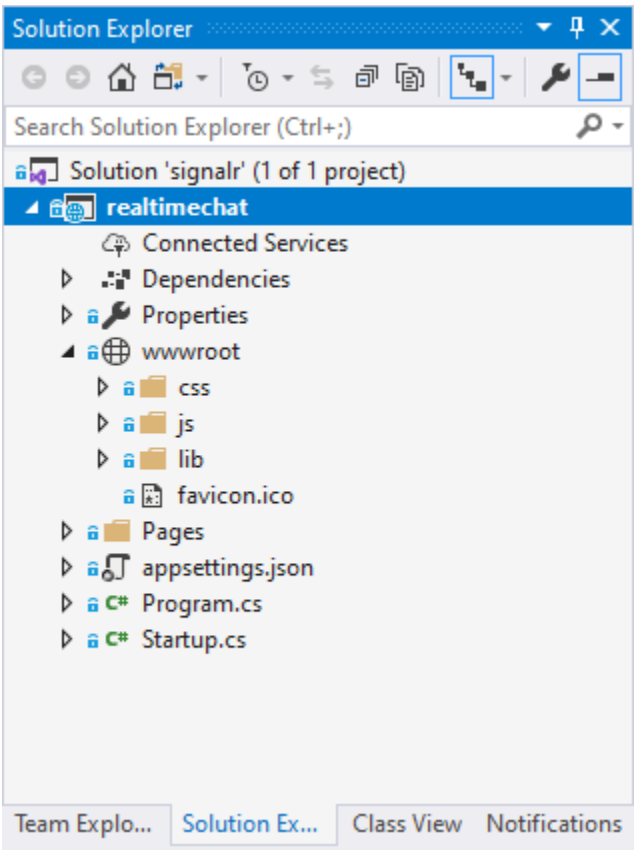


*Figure 18: Looking at the js Folder*

If you expand the **wwwroot** folder of your project, you will see a **js** folder. This is where we want to add the required JavaScript client library files.

***Note: The SignalR server library is automatically included in the ASP.NET Core 3.0 framework.***

Right-click your project and select **Add** > **Client-Side Library** from the context menu (Figure 19).

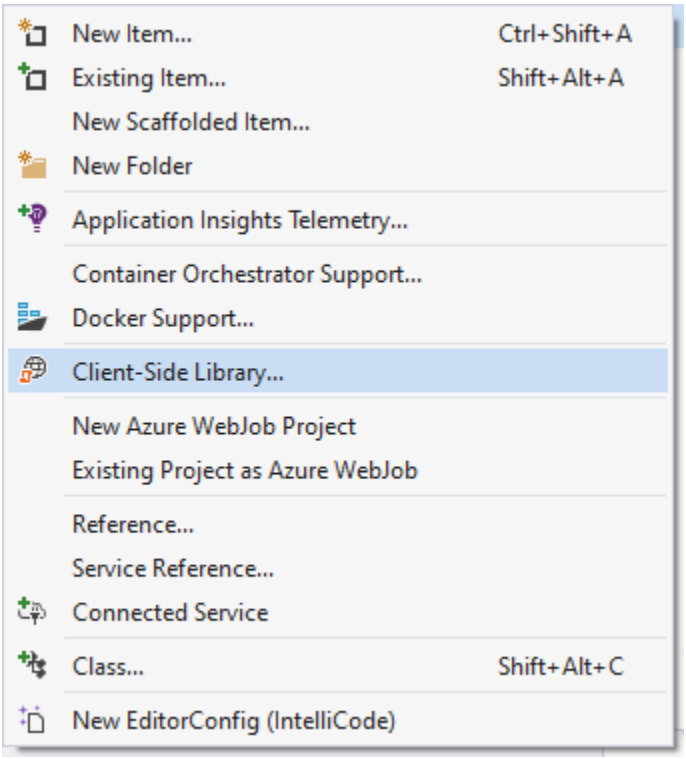


*Figure 19: Add a Client-Side Library*

From the **Add Client-Side Library** form that is displayed, ensure that you make the following selections:

- Set the **Provider** to **unpkg**.
- Enter **`@microsoft/signalr@latest`** in the **Library** field.
- Select the **Choose specific files** radio button, expand the tree to **dist/browser**, and select **signalr.js** and **signalr.min.js**.
- Change the **Target Location** path **to wwwroot/js/signalr**.

The target location is the folder in the project's wwwroot to which the js files will be copied (refer to Figure 20).

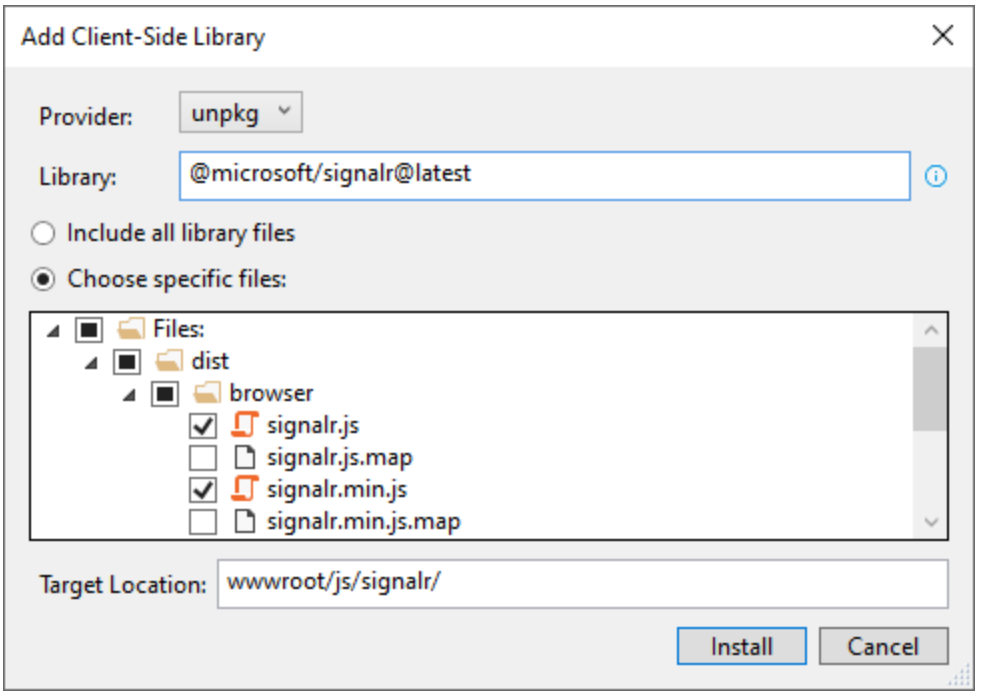


*Figure 20: Adding signalr.js Files*

Once you are ready, click **Install**. It might not immediately be obvious that Visual Studio is doing something, but after a few seconds, you will see the SignalR client library files added to your **wwwroot** folder.

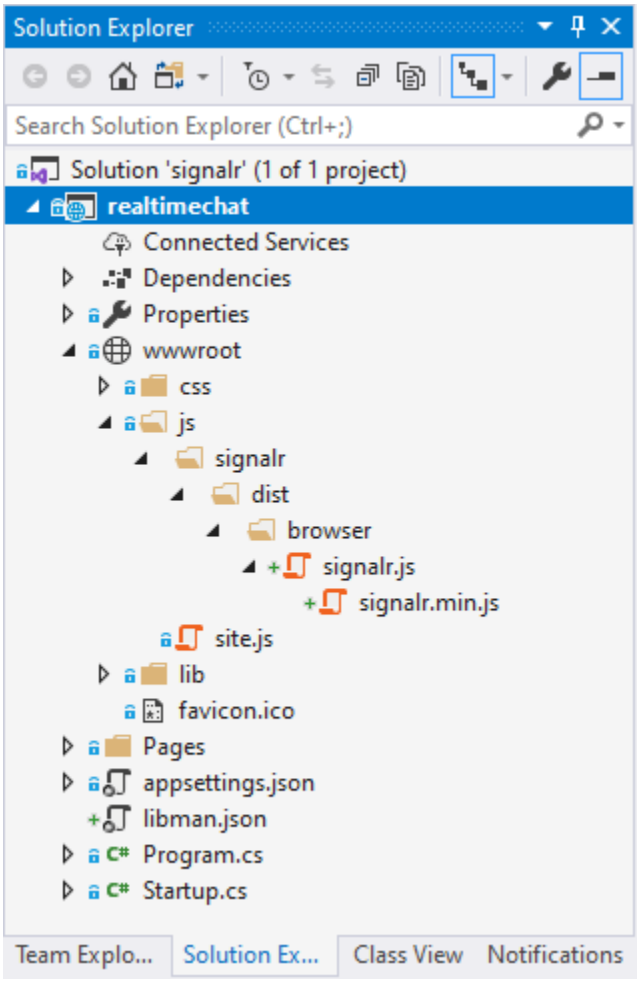


*Figure 21: Viewing the signalr.js Files Added*

After expanding the js/signalr folder, you will see the **signalr.js** and **signalr.min.js** files.

***Tip: The signalr.min.js is a minified file. When it comes to web browsers, you probably know that they don't care about the readability of code. Minification takes out all the whitespace and unnecessary carriage returns in your file. This creates a smaller file size, resulting in faster response times and lower bandwidth usage.***

We are now ready to start working on the SignalR hub. Let's do that next.

# Chapter 5 Creating the Hub and Configuring SignalR

The hub is what will handle client-server communication. Start by adding a folder called **Hubs** to your **realtimechat** project. Inside this folder, create a class called **`ChatHub`**. Inside this class, add the following code.

*Code Listing 5*

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

namespace realtimechat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task MessageSender(string user, string message)
        {
            await Clients.All.SendAsync("MessageReceiver", user,
message);
        }
    }
}
```

Let's pause here for a minute. Inside the **`MessageSender`** method, you will notice that we are using an object that invokes a method on all the clients connected to this hub. It does this via the **`All`** property.

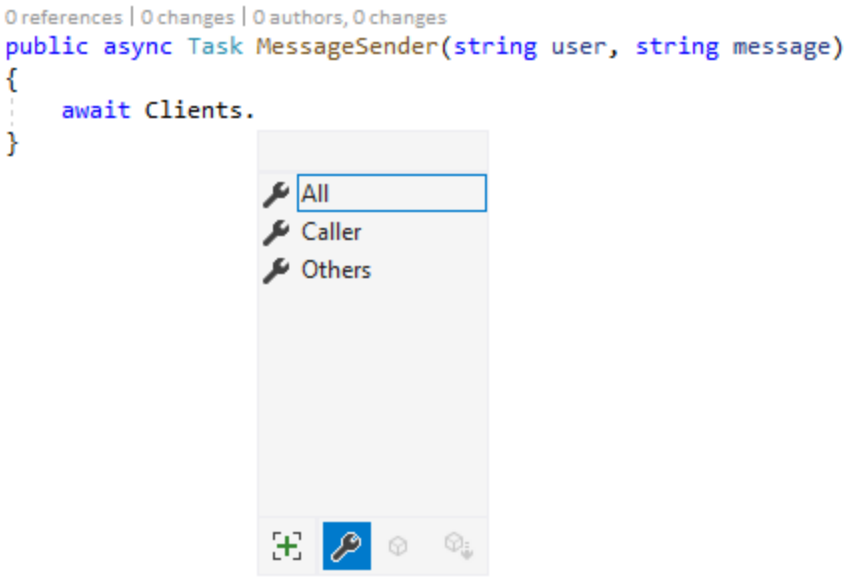


*Figure 22: Additional Clients Methods*

The **`Clients`** object implements the **`IHubCallerClients`** interface. This interface has the following properties:

- **`All`**—Invokes methods on all clients connected to the hub.
- **`Caller`**—Gets a caller to the connection that triggered the current invocation.
- **`Others`**—Gets a caller to all connections except the one that triggered the current invocation.

It also includes methods such as:

- **`AllExcept`**—Invokes methods on all clients connected to the hub, but excludes the specified connections passed as a read-only list of strings.
- **`Client`**—Invokes methods on a specific client connection.
- **`Clients`**—Invokes methods on the specified read-only list of client connections.
- **`Group`**—Invokes methods on all connections in the specified group.
- **`GroupExcept`**—Invokes methods on all connections in the specified group, but excludes the specified connections passed as a read-only list of strings.
- **`Groups`**—Invokes methods on all connections in the specified read-only list of groups.
- **`OthersInGroup`**—Gets a caller to all connections in the specified group, except the one which triggered the current invocation.
- **`User`**—Invokes methods on all connections associated with the specified user.
- **`Users`**—Invokes methods on all connections associated with the users specified in the read-only list of strings.

You will also notice that the **`ChatHub`** class inherits from the SignalR **`Hub`** class. This **`Hub`** base class is responsible for managing connections, groups, and messaging.

***Tip: To view the Hub metadata, place your cursor on the Hub class name and press F12.***

Having a look at the metadata for the **`Hub`** class, we can see the following properties:

- **Clients**—Gets or sets an object that is used to invoke methods on the clients that are connected to this hub. You can see this in Code Listing 1.
- **Context**—Allows you to get or set the hub caller context. (I discuss this in more detail in the section on Hubs.)
- **Groups**—Allows you to get or set the group manager.

The **Hub** class also has the following properties:

- **OnConnectedAsync**—This is called when a new connection is established with the hub and returns a task representing the asynchronous connect.
- **OnDisconnectedAsync**—This is called when the connection to the hub is terminated and returns a task representing the asynchronous disconnect.
- **Dispose**—The Hub class implements **IDisposable** and, therefore, will be able to dispose of all the resources currently in use by this hub instance.

Referring back to the code in Code Listing 5, the **MessageSender** method is therefore called by a connected client and will send a message to all the clients.

## Configuring SignalR

Let's focus our attention on the **Startup** class of the project. Start by adding the **using realtimechat.Hubs** statement to the **Startup** class.

The next line of code we need to add is to the **ConfigureServices** method, as shown in Code Listing 6.

*Code Listing 6*

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    services.AddSignalR();
}
```

In the **Configure** method, add the following endpoint: **endpoints.MapHub<ChatHub>("/chatHub")**. Your code should look like Code Listing 7.

*Code Listing 7*

```
app.UseEndpoints(endpoints =>
{
     endpoints.MapRazorPages();
     endpoints.MapHub<ChatHub>("/chatHub");
});
```

If you have added everything correctly, your **Startup** class will look as in Code Listing 8.

*Code Listing 8*

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using realtimechat.Hubs;

namespace realtimechat
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddRazorPages();
            services.AddSignalR();
        }

        //Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
```

```
            {
                endpoints.MapRazorPages();
                endpoints.MapHub<ChatHub>("/chatHub");
            });
        }
    }
}
```

Let's have a look at the **`Startup`** class in more detail.

## The Startup class

If you have been a .NET developer for any length of time, you will know that there is a requirement in .NET for command-line applications. That requirement is to have a **`static void Main`** method, typically in a class called **`Program`**.

Opening up the **`Program`** class of the **`realtimechat`** project, you will see the code shown in Code Listing 9.

*Code Listing 9*

```
namespace realtimechat
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Here you will see the **`public static void Main`** method mentioned previously. This is the entry point (where execution will start) when the **`realtimechat`** application starts running. The **`Main`** method is used to create a host builder, specifically a default builder.

A builder, in this case, knows how to create a web host. The default builder also sets up some of the default services behind the scenes. In Code Listing 9, you will also see that we are telling this default builder to configure our application using the **`Startup`** class by adding the line of code **`webBuilder.UseStartup<Startup>()`**.

In Code Listing 10, in the **`Main`** method, we **`Build()`** the builder. This gives us a web host instance that knows how to listen for connections and how to process and receive HTTP messages. Finally, we tell the web host to **`Run()`**.

*Code Listing 10*

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}
```

Once we call **`Run()`**, everything that happens in the application is determined by the code that is contained in the **`Startup`** class.

If you place your mouse cursor on the **`Startup`** in **`UseStartup<Startup>`** and press **F12**, you will jump to the **`Startup`** class we added our code to earlier, as outlined in Code Listing 8. ASP.NET Core now knows that it's going to instantiate the **`Startup`** class and invoke two methods.

The first method is **`ConfigureServices`**. The **`ConfigureServices`** method will allow us to add our own custom services in ASP.NET Core. This will then inject those services into pages, controllers, and wherever else we might need them.

The second is the **`Configure`** method. This will determine what middleware it will execute for every incoming HTTP message.

***Note: The request pipeline comprises of a series of middleware components. Each middleware component performs some operation on the `HttpContext`, and then invokes the next middleware in the pipeline or terminates the request.***

You can also create custom middleware components, but this is beyond the scope of this book.

# Chapter 6  Creating the Client Application

In the **realtimechat** project, expand the **Pages** folder and edit the **Index.cshtml** file.

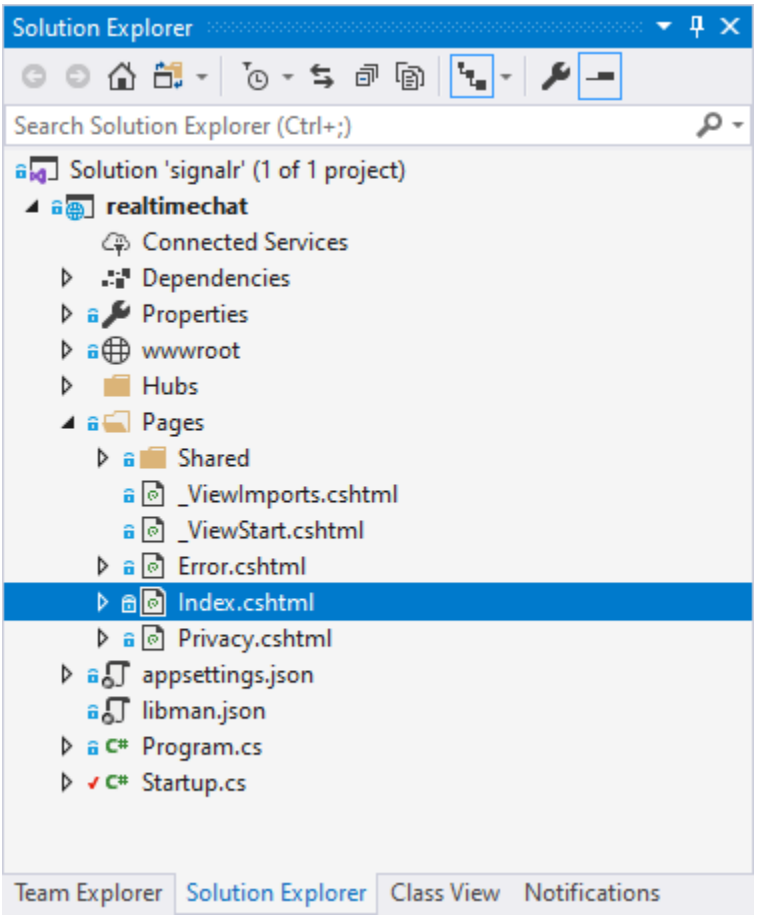


*Figure 23:The Index.cshtml File*

You will see the following default HTML code.

*Code Listing 11*

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
```

```
    <p>Learn about <a
href="https://docs.microsoft.com/aspnet/core">building Web apps with
ASP.NET Core</a>.</p>
</div>
```

Replace the code in the **Index.cshtml** file with the code in Code Listing 12.

*Code Listing 12*

```
@page
<form>
    <div class="form-group">
        <label for="userInput">User</label>
        <input class="form-control" placeholder="Enter your name"
type="text" id="userInput" />
    </div>
    <div class="form-group">
        <label for="messageInput">Message</label>
        <input class="form-control" placeholder="Type your message"
type="text" id="messageInput" />
    </div>
    <div class="form-group">
        <input class="btn btn-primary" type="button" id="sendButton"
value="Send Message" />
    </div>
</form>


<div class="row">
    <div class="col-12">
        <hr />
    </div>
</div>
<div class="row">
    <div class="col-6">
        <ul class="list-group" id="messagesList"></ul>
    </div>
</div>

<script src="~/js/signalr/dist/browser/signalr.js"></script>

<script src="~/js/chat.js"></script>
```

The code in Code Listing 12 simply adds text boxes for the user's name and the message that they want to send. It also includes a send button. There is also a list called **`messagesList`** that will display all the messages received from the SignalR hub.

At the bottom of the code, you will see script references to **signalr.js** and **chat.js**. You will recall that we added the **signalr.js** file when we added the client-side libraries to our project (Figure 21). We have not, however, added the **chat.js** file to our project yet.

To do this, right-click your js folder and add a new JavaScript file, as illustrated in Figure 24.

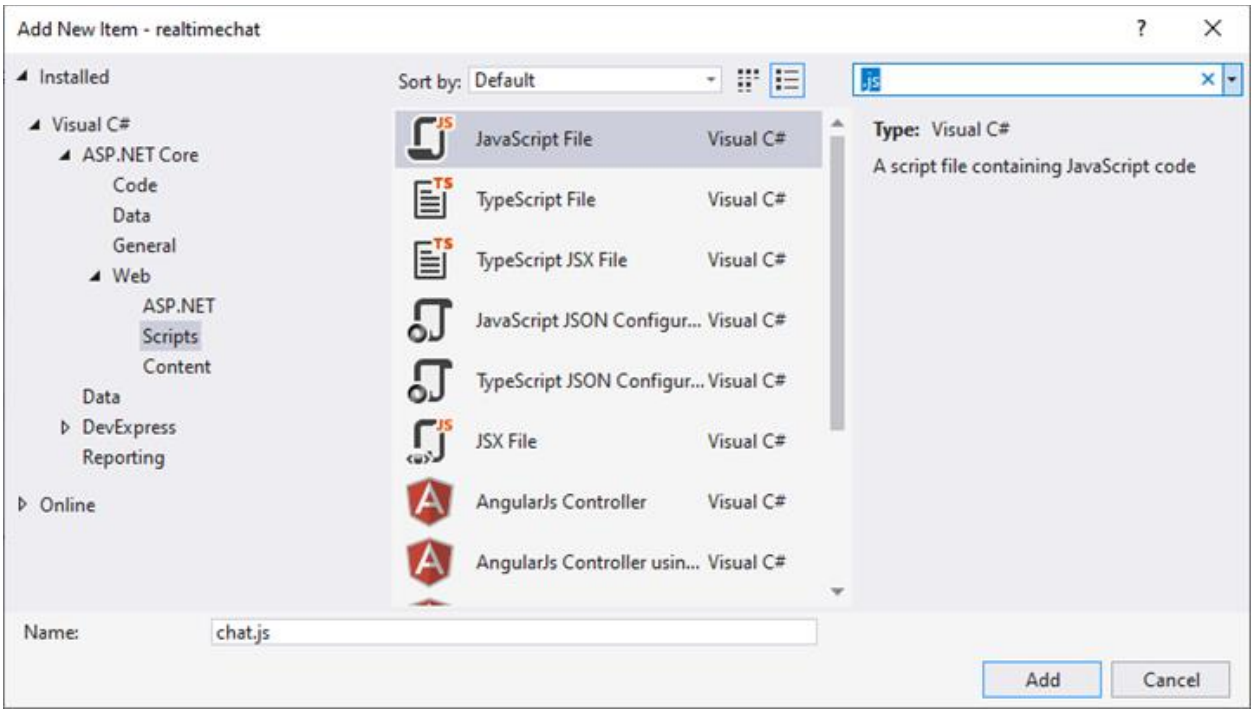


*Figure 24: Add New Item*

Call the file **chat.js** and click **Add**. Your solution should now look as illustrated in Figure 25.

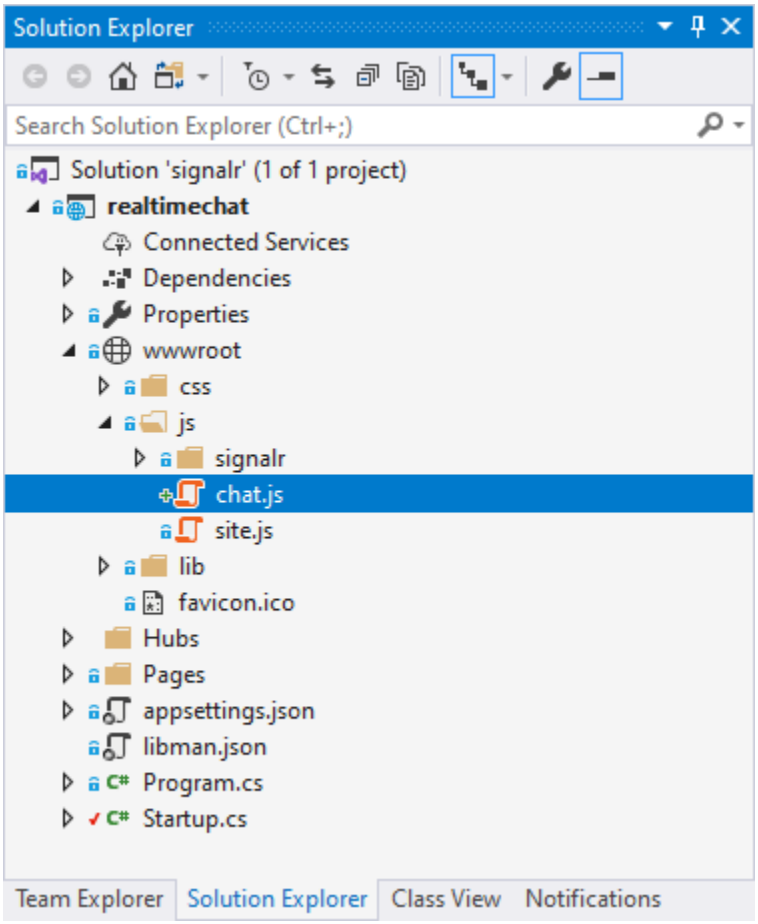


*Figure 25: Added chat.js File*

After adding your **chat.js** file, you should see the blank code editor. Add the following JavaScript code to your **chat.js** file.

*Code Listing 13*

```
"use strict";

var connection = new
signalR.HubConnectionBuilder().withUrl("/chatHub").build();

//Disable send button until connection is established.
document.getElementById("sendButton").disabled = true;

connection.on("MessageReceiver", function (user, message) {
    var msg = message.replace(/&/g, "&amp;").replace(/</g,
"&lt;").replace(/>/g, "&gt;");
    var encodedMsg = user + ": " + msg;
    var li = document.createElement("li");

    var currentUser = document.getElementById("userInput").value;
```

```
    if (currentUser === user) {
        li.className = "list-group-item list-group-item-primary";
    }
    else {
        li.className = "list-group-item list-group-item-success";
    }

    li.textContent = encodedMsg;
    document.getElementById("messagesList").appendChild(li);
});

connection.start().then(function () {
    document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function
(event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
    connection.invoke("MessageSender", user, message).catch(function
(err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

Let's have a closer look at this JavaScript file. Essentially, it does three things:

- It creates and starts a connection.
- It creates an event handler for the Submit button that sends messages to the hub.
- It adds an event handler to the connection object that receives messages from the hub and adds those received messages to the message list.

***Note: I will not go into detail regarding the chat.js file's code. Later on, we will create a SignalR charting app. This will use a similar JavaScript file called charting.js. I will expand on the JavaScript in the file when we discuss the real-time charting application.***

# Chapter 7  Running the Application

You have now completed all the code needed for your real-time chat application. Run your application in Visual Studio by pressing **Ctrl+F5**, or if you are using Visual Studio for Mac, select **Run > Start Without Debugging**. If you are using Visual Studio Code, type the following command in the integrated terminal.

*Code Listing 14*

```
dotnet watch run -p realtimechat.csproj
```

Once your application starts in your browser, copy the URL, open another instance of the browser, and paste the URL in the address bar.

Enter your username and type a message, as illustrated in Figure 26. Select **Send Message**.

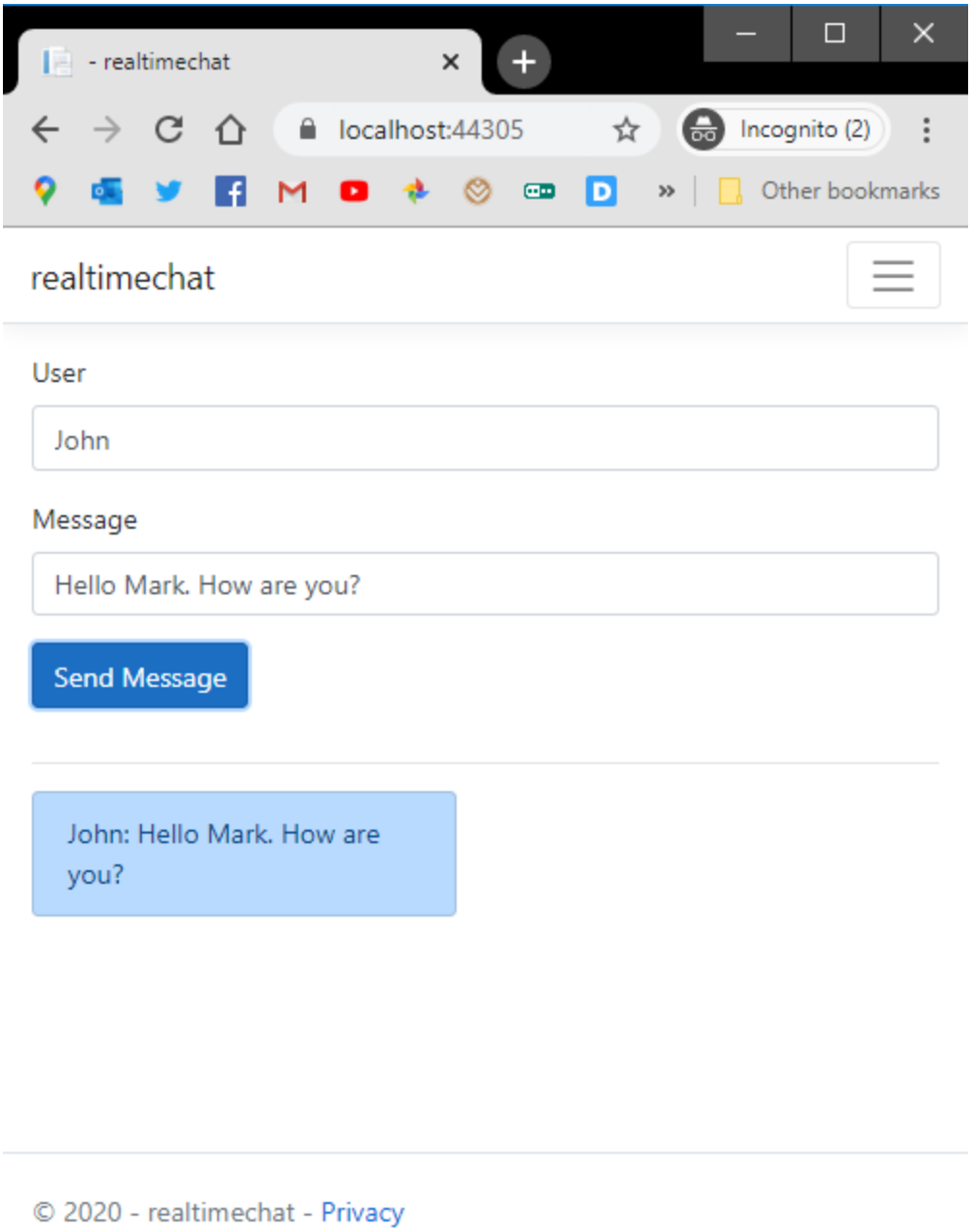


*Figure 26: Browser 1 Message*

When you click the **Send Message** button, the message is immediately displayed in the list at the bottom of the browser page on browser 1. If you look at browser 2, illustrated in Figure 27, you will see the same message displayed in that browser.

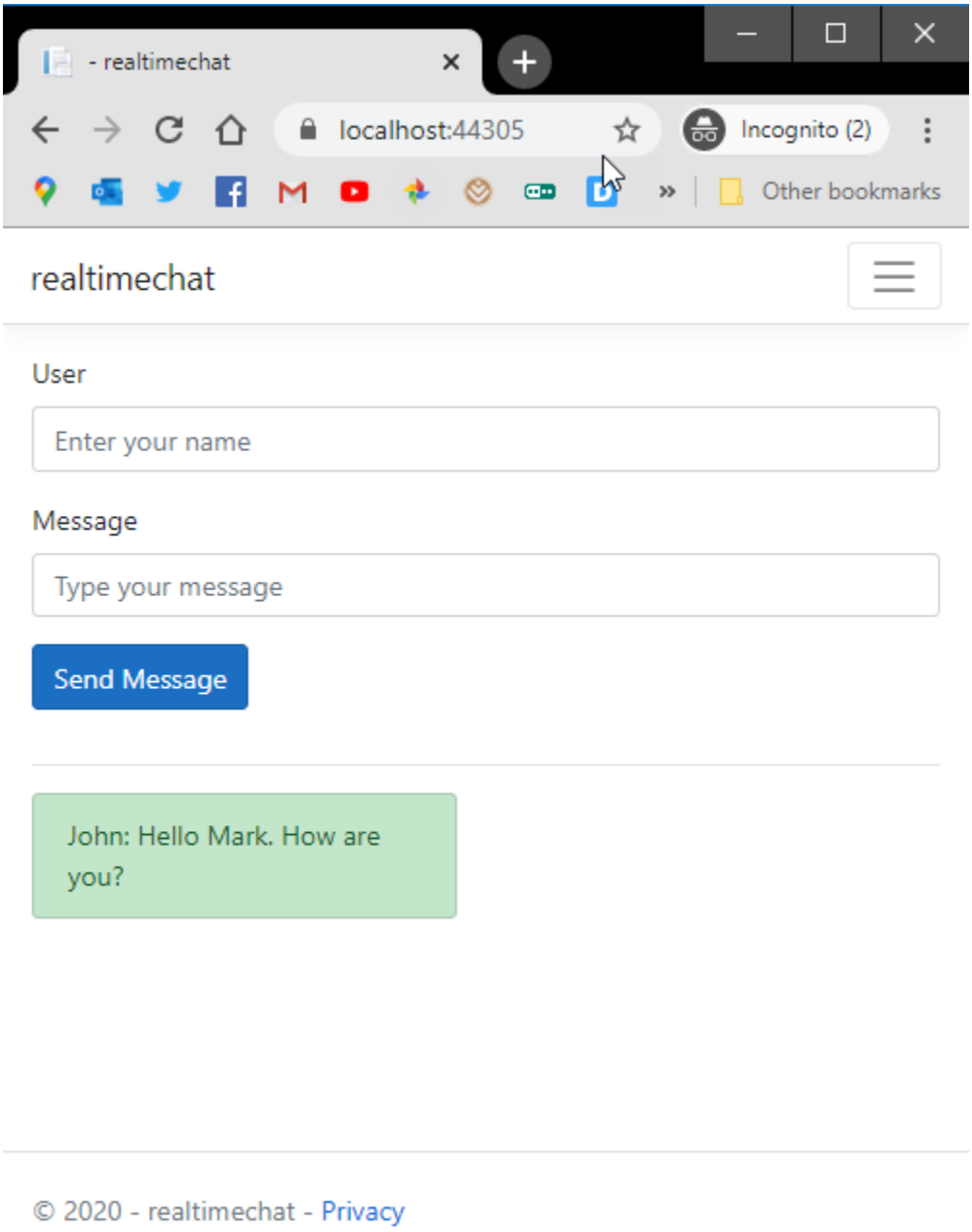


*Figure 27: Browser 2 Message Received*

In browser 2, add the username **Mark**, type a reply for John in the **Message** field, and select **Send Message**.

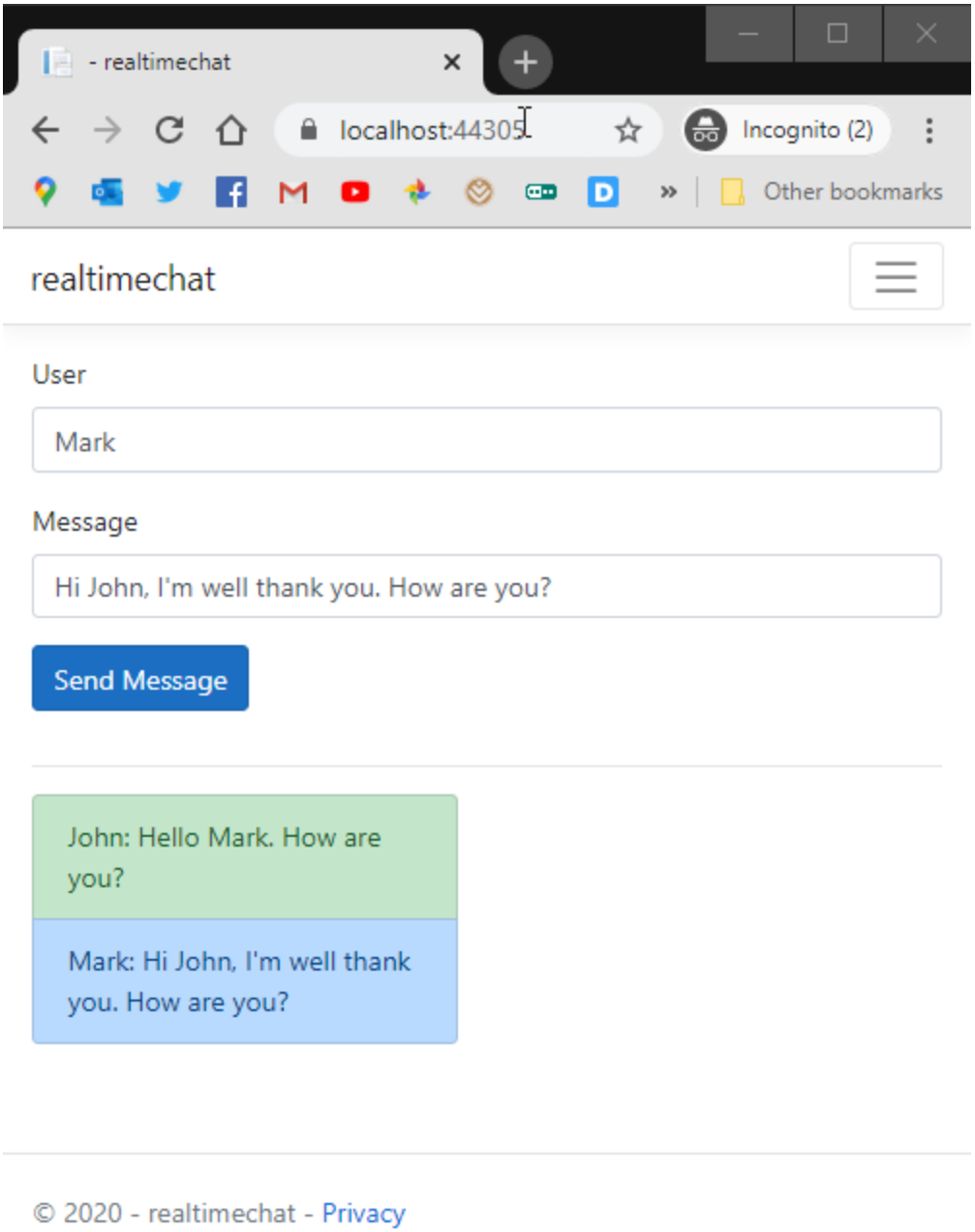


*Figure 28: Browser 2 Reply*

As seen in Figure 28, the reply is displayed in the list, and on browser 1 (John's browser), as seen in Figure 29.

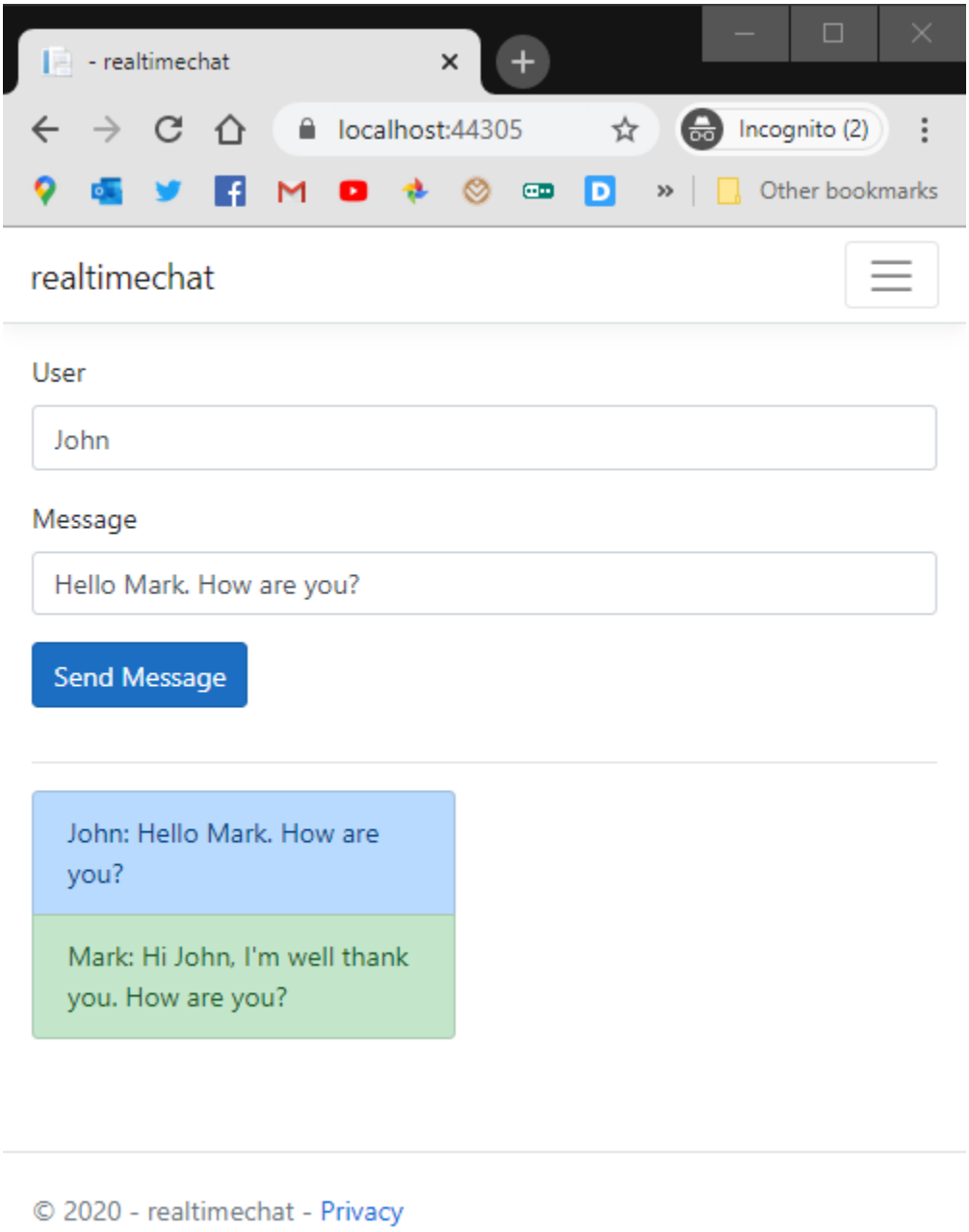


*Figure 29: Browser 1 Reply from Mark Received*

Messages from other users are green, while messages that you type are blue. This logic is contained in the **chat.js** file that adds a class to the **`<li>`** element based on the current username.

# Chapter 8  The Problem with SendAsync

There is a slight drawback to using **SendAsync** in the hub. This is a bug that can easily slip in and cause a lot of frustration. You may have noticed it way back in Code Listing 5, when we created the hub. Have a look at Code Listing 15.

*Code Listing 15*

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

namespace realtimechat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task MessageSender(string user, string message)
        {
            await Clients.All.SendAsync("MessageReciever", user,
message);
        }
    }
}
```

Compare this to Code Listing 16.

*Code Listing 16*

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

namespace realtimechat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task MessageSender(string user, string message)
        {
            await Clients.All.SendAsync("MessageReceiver", user,
message);
        }
    }
}
```

Do you see the issue? The method being specified in Code Listing 15 is spelled incorrectly. Instead of **`MessageReceiver`**, the developer typed in **`MessageReciever`**. It's an easy mistake to make because the **`SendAsync`** method relies on a string to specify the client method name to be called. Compounding this issue, the fact that this is a misspelled string leaves your application open to runtime errors, which aren't caught at compile time.

The solution to this is to use strongly typed hubs. This means we strongly type the **`Hub`** with **`Hub<T>`**. To accomplish this, we need to extract the **`ChatHub`** client method to an interface. We will call this interface **`IChatClient`**.

Right-click your **Hubs** folder in your solution and add a new interface called **`IChatClient`**. When you have added your interface, your solution should look like Figure 30.

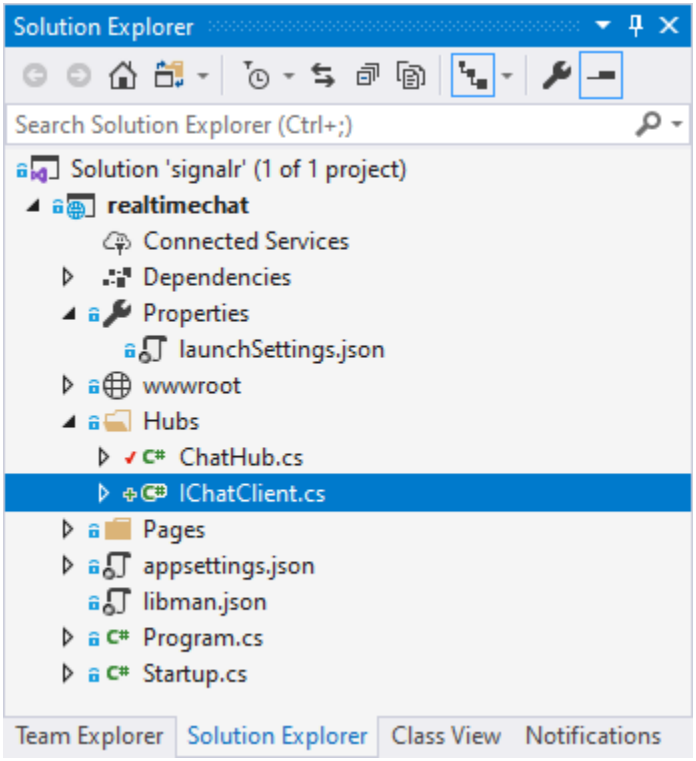


*Figure 30: IChatClient Interface Added*

To this interface, add the following code.

*Code Listing 17*

```
using System.Threading.Tasks;

namespace realtimechat.Hubs
{
    public interface IChatClient
    {
        Task MessageReceiver(string user, string message);
    }
```

```
}
```

Now go to your **ChatHub** class and modify your code to use **Hub<T>** where **T** is your **IChatClient** interface. Replace the **SendAsync** method to use the method defined in your **IChatClient** interface. After completing all this, your **ChatHub** code will look like Code Listing 18.

*Code Listing 18*

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

namespace realtimechat.Hubs
{
    public class ChatHub : Hub<IChatClient>
    {
        public async Task MessageSender(string user, string message)
        {
            await Clients.All.MessageReceiver(user, message);
        }
    }
}
```

We can see that by using **Hub<IChatClient>**, we allow compile-time checking of the client methods. This is because the **Hub<T>** can only provide access to the methods as explicitly defined in the **IChatClient** interface.

# Chapter 9  Creating Real-Time Charts

You should now have an idea of the power that SignalR gives developers when they're creating real-time web applications. You might be wondering how easy it would be to create another type of real-time application.

Let us have a look at something more suited to business applications. We will create a real-time ASP.NET Core web application that displays a line chart.

Create a new ASP.NET Core 3.0 web application as outlined in the previous sections, "Project Creation and Setup" and "Add the Required SignalR Client Library." Call the new project **realtimechart**.

Once you have added your ASP.NET Core 3.0 web application to your solution, your **Solution Explorer** window will look as shown in Figure 31.

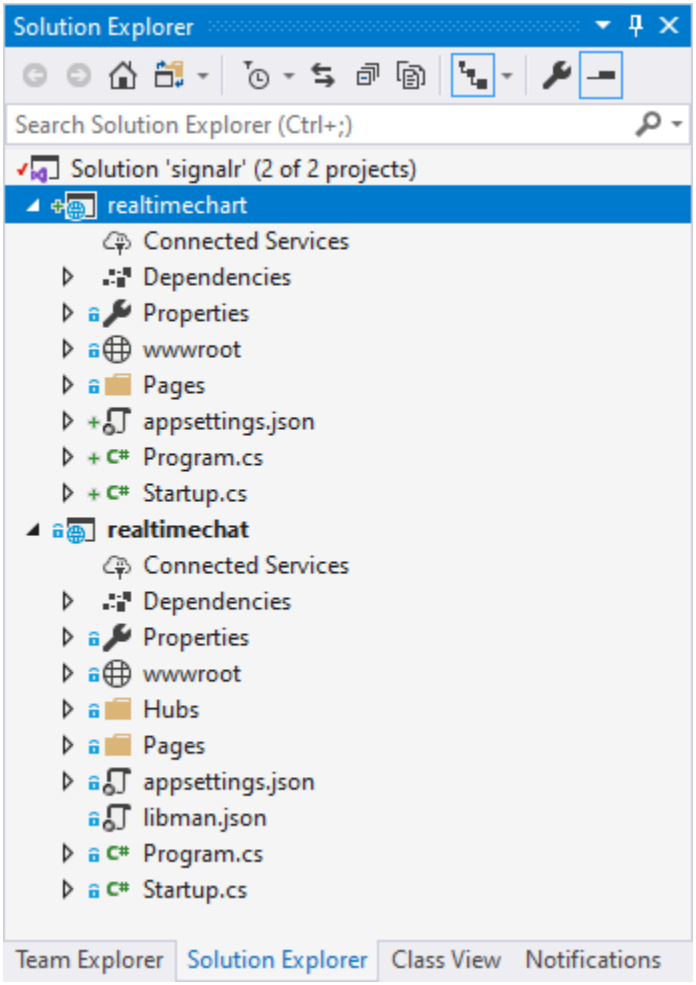


*Figure 31: Adding Real-Time Charting Application*

With your new project added to your solution, we can now begin to add the logic needed to display the real-time chart.

***Note: From here on, the book assumes that you have added the required SignalR Client Library to your new project, as illustrated previously in the book.***

Real-time charting can be very beneficial in many applications. Think of a real-time KPI (key performance indicator) dashboard of sales figures, a stock market price ticker, a shared calendar scheduler, or even a doctor in/out notification dashboard for a hospital ward.

# Chapter 10  Creating the Chart Hub

Start by creating a new folder called Hubs in your **realtimechart** application. Inside this folder, create a class called **ChartHub**. Your solution will look as illustrated in Figure 32.

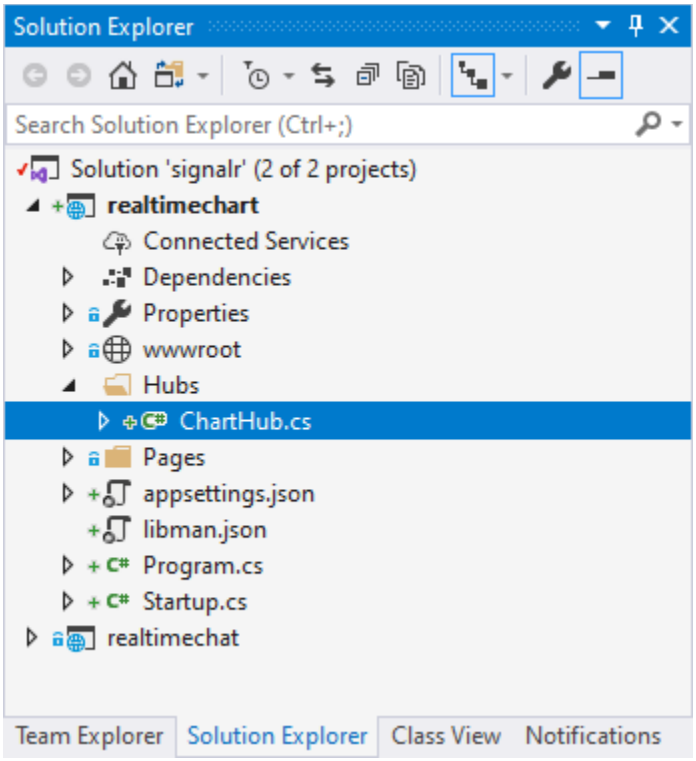


*Figure 32: Adding the Chart Hub*

The code of your new **`ChartHub`** class will look like Code Listing 19. We have not added any logic to it regarding SignalR.

*Code Listing 19*

```
namespace realtimechart.Hubs
{
    public class ChartHub
    {
    }
}
```

The next thing we are going to do is to add an interface to our project. For simplicity's sake, we will just add the interface to the **Hubs** folder.

Right-click your **Hubs** folder and select **Add > New Item** from the context menu. Select an interface and call your interface IChartClient. Click **Add**.

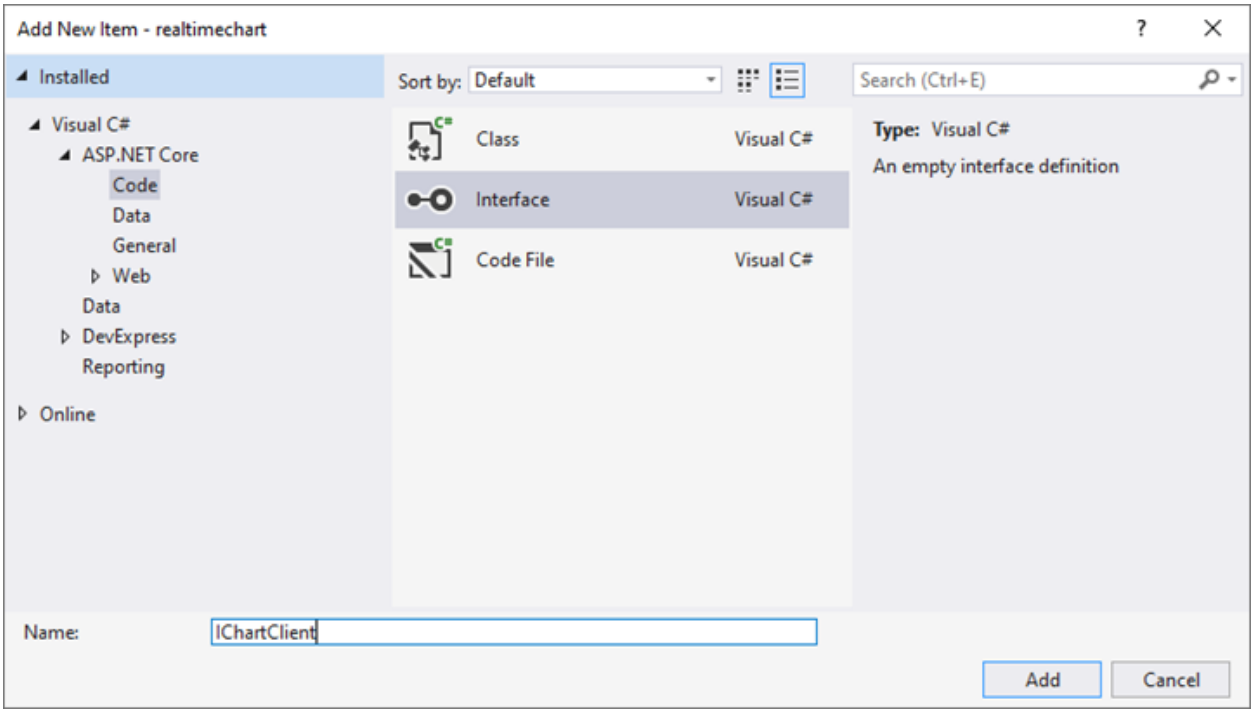


*Figure 33: Adding IChartClient Interface*

Once your interface has been added, change the code to look like Code Listing 20.

*Code Listing 20*

```
using System.Threading.Tasks;

namespace realtimechart.Hubs
{
    public interface IChartClient
    {
        Task ValueReceiver(double chartValue);
    }
}
```

We can now go ahead and flesh out our **`ChartHub`** class using the **`IChartClient`** interface. Modify your **`ChartHub`** class as illustrated in Code Listing 21.

*Code Listing 21*

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

namespace realtimechart.Hubs
```

```
{
    public class ChartHub : Hub<IChartClient>
    {
        public async Task ValueSender(double chartValue)
        {
            await Clients.All.ValueReceiver(chartValue);
        }
    }
}
```

At this stage, your solution should look as illustrated in Figure 34. You should have the **`ChartHub`** class with the **`IChartClient`** interface in the **Hubs** folder.

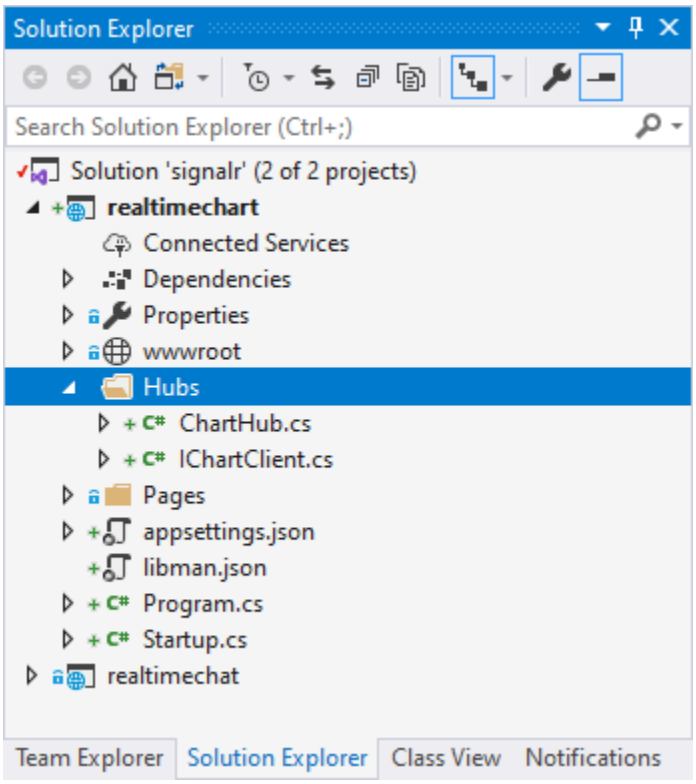


*Figure 34:The Solution So Far*

We now need to go and configure the **`Startup`** class. Go ahead and open your **`Startup`** class. Add the SignalR service to your **`ConfigureServices`** method, as seen in Code Listing 22.

*Code Listing 22*

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    services.AddSignalR();
```

```
}
```

In the **Configure** method, go ahead and add the following endpoint: **endpoints.MapHub<ChartHub>("/chartHub")**. Your code should look like Code Listing 23. This maps the path **/chartHub** to the **ChartHub** class.

*Code Listing 23*

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
    endpoints.MapHub<ChartHub>("/chartHub");
});
```

After adding everything to your **Startup** class, your code should look like Code Listing 24.

*Code Listing 24*

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using realtimechart.Hubs;

namespace realtimechart
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddRazorPages();
            services.AddSignalR();
        }

        // Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
```

```
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapRazorPages();
                endpoints.MapHub<ChartHub>("/chartHub");
            });
        }
    }
}
```

Once you have finished configuring your **`Startup`** class, it's time to create the client application, which we'll do in the next section.

# Chapter 11 Creating the Chart Client Application

In the **realtimechart** project, expand the **Pages** folder and edit the **Index.cshtml** file.

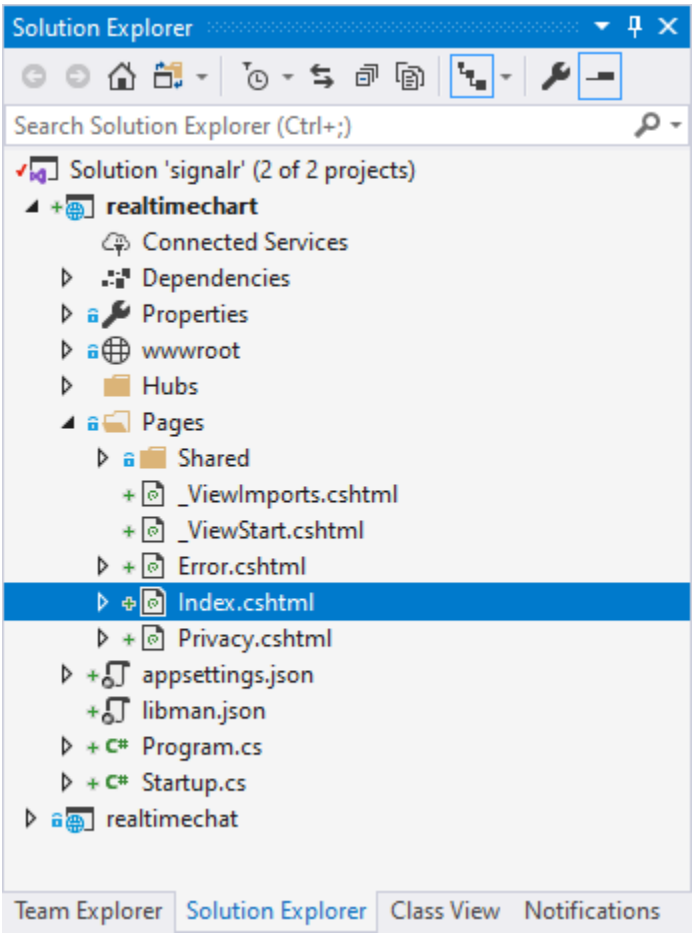


*Figure 35: Index Page for Realtime Chart*

You will see the following default HTML code, as shown in Code Listing 25.

*Code Listing 25*

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
```

```
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a
href="https://docs.microsoft.com/aspnet/core">building Web apps with
ASP.NET Core</a>.</p>
</div>
```

Replace the code in the **Index.cshtml** file with the code in Code Listing 26.

*Code Listing 26*

```
@page
<form>
    <div class="form-group">
        <label for="messageInput">Value</label>
        <input class="form-control" placeholder="Enter a value"
type="text" id="valueInput" />
    </div>
    <div class="form-group">
        <input class="btn btn-primary" type="button" id="sendButton"
value="Enter Value" />
    </div>
</form>

<div class="row">
    <div class="col-12">
        <hr />
    </div>
</div>

<div id="chartArea">
</div>


@section Scripts
{
    <script src="~/js/signalr/dist/browser/signalr.js"></script>
    <script src="~/js/charting.js"></script>
}
```

The code in Code Listing 26 simply adds a text box and a Submit button to your page. There is also a **`div`** called **`chartArea`** that will display all the chart values received from the SignalR hub and plot them on the chart.

At the bottom of the code, you will see script references to **signalr.js** and **charting.js**. I have added these to a **`Scripts`** section. We have not added the **charting.js** file to our project yet. We will do this in a minute. First, I want to add the required references to **Chartist** to generate the line chart on our page.

***Note: You can read more about Chartist here.***

Expand the **Pages** folder in your solution, and then expand the **Shared** folder.

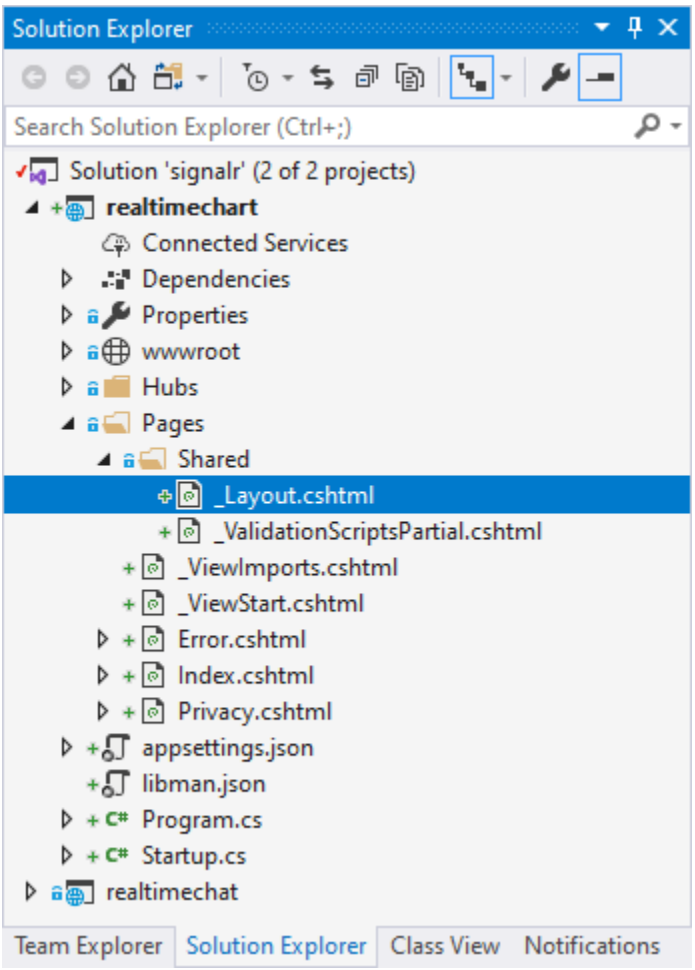


*Figure 36: Edit the _Layout Page*

Inside the **Shared** folder, you will see the **_Layout.cshtml** page.

***Tip: For all the ASP.NET Web Pages devs out there, think of this as a Master Page.***

In the **<head>** section of the **_Layout.cshtml** page, add a reference to the **chartist.min.css** file served from the CDNJS (content delivery network – JavaScript) on GitHub, as seen in Code Listing 27.

*Code Listing 27*

```
<link rel="stylesheet"
href="//cdn.jsdelivr.net/chartist.js/latest/chartist.min.css">
```

At the bottom of the _**Layout.cshtml** page, add a reference to the **chartist.min.js** file served from the CDNJS in GitHub, as seen in Code Listing 28.

*Code Listing 28*

```
<script
src="//cdn.jsdelivr.net/chartist.js/latest/chartist.min.js"></script>
```

You can find the js and css files here. The complete _**Layout.cshtml** page will look like Code Listing 29.

*Code Listing 29*

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <title>@ViewData["Title"] - realtimechart</title>
    <link rel="stylesheet"
href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" />
    <link rel="stylesheet"
href="//cdn.jsdelivr.net/chartist.js/latest/chartist.min.css">
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-
page="/Index">realtimechart</a>
                <button class="navbar-toggler" type="button" data-
toggle="collapse" data-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area=""
asp-page="/Index">Home</a>
                        </li>
                        <li class="nav-item">
```

```
                            <a class="nav-link text-dark" asp-area=""
asp-page="/Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2020 - realtimechart - <a asp-area="" asp-
page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script
src="//cdn.jsdelivr.net/chartist.js/latest/chartist.min.js"></script>
    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script
src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

You will notice the **`@RenderSection("Scripts", required: false)`** at the bottom of this page. It is from here that the **`Scripts`** section mentioned in Code Listing 26 is called.

We are now ready to add our **charting.js** file to the project. To do this, right-click your **js** folder (located in wwwroot) and add a new JavaScript file, as illustrated in Figure 37.

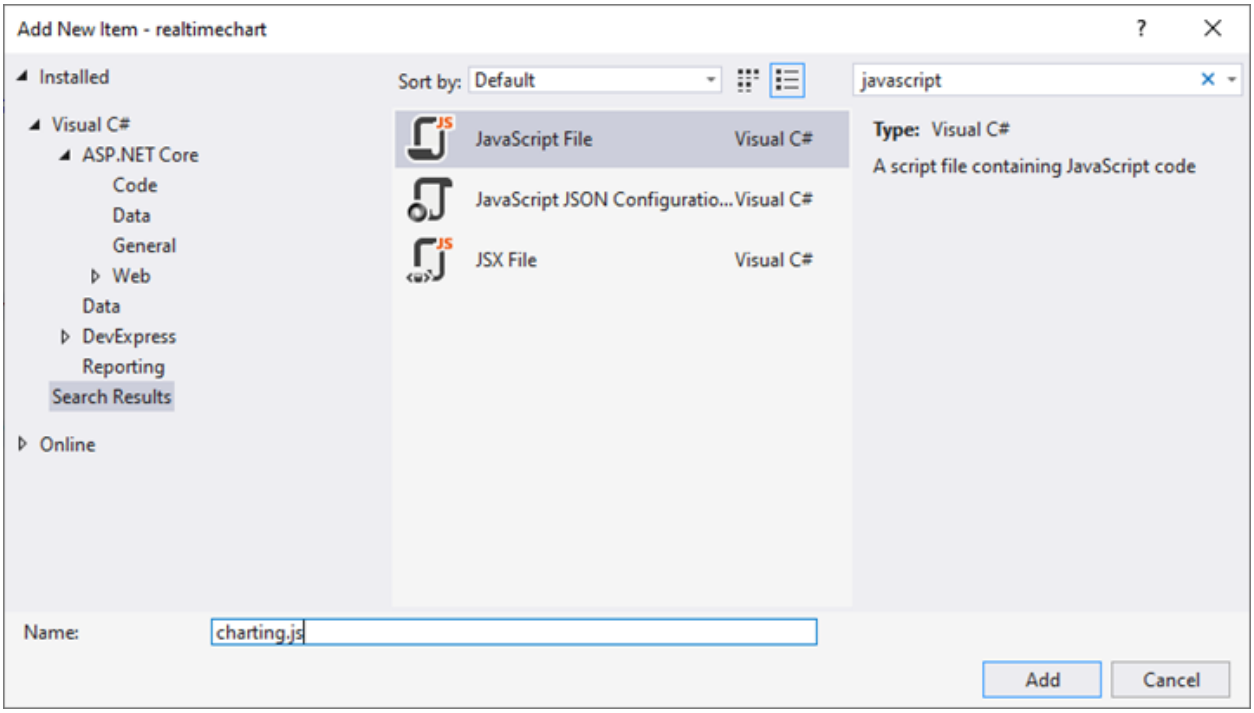


*Figure 37:Adding a New .js File*

Call the file **charting.js** and click **Add**. Your solution should now look as illustrated in Figure 38.

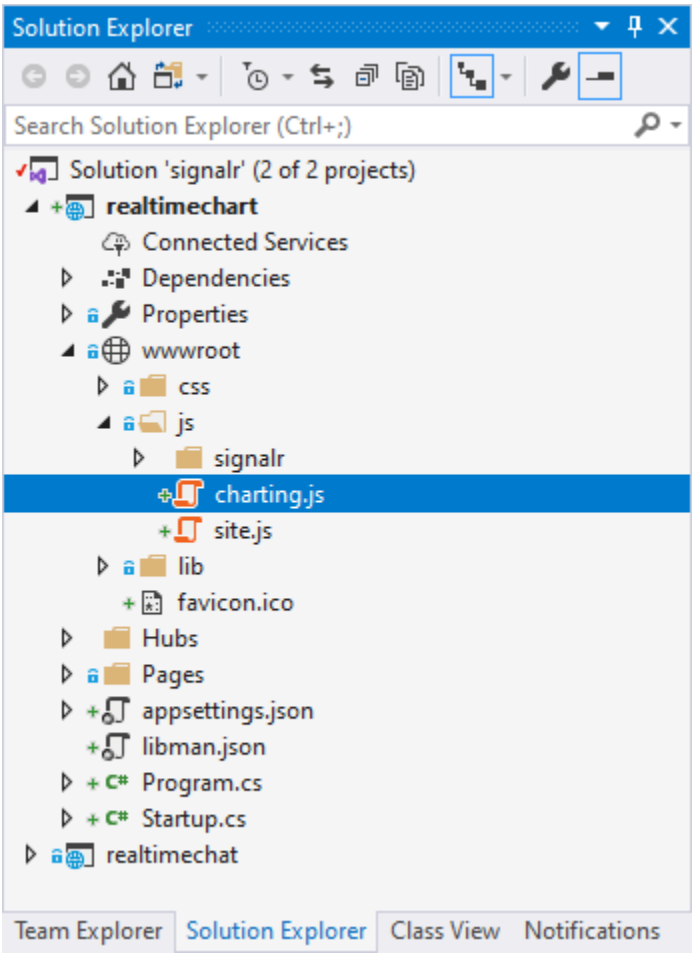


*Figure 38: Added charting.js File to Project*

After adding your **charting.js** file, you should see the blank code editor. Add the following JavaScript code to your **charting.js** file.

*Code Listing 30*

```
"use strict";

var lineChart = new Chartist.Line('#chartArea', {
    labels: [],
    series: [[]]
},
    {
        low: 0,
        showArea: true
    });

var connection = new
signalR.HubConnectionBuilder().withUrl("/chartHub").build();
```

```
//Disable send button until connection is established.
document.getElementById("sendButton").disabled = true;

connection.on("ValueReceiver", function (chartValue) {
    if (chartValue && !isNaN(chartValue)) {
        lineChart.data.series[0].push(chartValue);
        lineChart.update();

        document.getElementById("valueInput").value = "";
        document.getElementById("valueInput").focus();
    }
});

connection.start().then(function () {
    document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function
(event) {
    var strValue = document.getElementById("valueInput").value;

    var chartValue = parseFloat(strValue);

    connection.invoke("ValueSender", chartValue).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});

$('#valueInput').keypress(function (e) {
    var key = e.which;
    if (key === 13)  // the enter key code.
    {
        $('#sendButton').click();
        return false;
    }
});
```

You will notice that I have not added any validation of inputs in this JavaScript code. If you want to validate the user input, I suggest doing it in the **charting.js** file.

Let's have a look at some of the portions of this JavaScript code next.

# Chapter 12 The JavaScript Code Explained

The **chat.js** file's code we created in the chat application (Code Listing 13) functions similarly to the **charting.js** file's code. I would suggest opening up both files and comparing them side by side to see how they are similar and where they differ.

In the first section (Code Listing 31) in the **charting.js** code, we create a new line chart object called **`lineChart`** using **`Chartist`**. You will notice that it references the **`<div>`** called **`chartArea`** by using its jQuery #id selector, **`#chartArea`**.

*Code Listing 31*

```
var lineChart = new Chartist.Line('#chartArea', {
    labels: [],
    series: [[]]
},
{
    low: 0,
    showArea: true
});
```

Next, we need to build a new SignalR **`HubConnection`**, as seen in Code Listing 32. You will notice that the **`withUrl("/chartHub")`** matches the **`endpoints.MapHub<ChartHub>("/chartHub")`** specified in the **`Configure`** method of the **Startup.cs** file. This was added to the **Startup.cs** file to map the incoming requests with the specified path to the specified SignalR Hub.

*Code Listing 32*

```
var connection = new
signalR.HubConnectionBuilder().withUrl("/chartHub").build();
```

The next section of code in Code Listing 33 adds an event listener on the **`sendButton`** click. This will run the code every time you click the button to add a value. It gets the text value and ensures it is parsed to a float.

***Note: You might want to do some more validation here, but for this demo, I didn't add any.***

The code then invokes the method on the server called **`ValueSender`**. This is the method we created in our **`ChartHub`** hub.

*Code Listing 33*

```
document.getElementById("sendButton").addEventListener("click", function
(event) {
    var strValue = document.getElementById("valueInput").value;

    var chartValue = parseFloat(strValue);

    connection.invoke("ValueSender", chartValue).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

It might also be a good idea to add an event listener on the Enter key just after the user has added a value in the **valueInput** text box. Here in Code Listing 34, I'm using the jQuery #id selector again.

If the key being pressed equals 13, then the Enter key has been pressed. I then call the button click event of the **Enter Value** button, called **sendButton**.

*Code Listing 34*

```
$('#valueInput').keypress(function (e) {
    var key = e.which;
    if (key === 13)  // the enter key code.
    {
        $('#sendButton').click();
        return false;
    }
});
```

When the **ValueSender** method on the **ChartHub** has been invoked, it in turn calls the **ValueReceiver** method on all connected clients (Code Listing 35). This checks to see if the chart value passed to it is a number and has a value.

It then pushes this value onto the **lineChart** object and updates the chart. Then it clears the text box and sets the focus to that text box, ready to receive the next value.

*Code Listing 35*

```
connection.on("ValueReceiver", function (chartValue) {
    if (chartValue && !isNaN(chartValue)) {
        lineChart.data.series[0].push(chartValue);
        lineChart.update();

        document.getElementById("valueInput").value = "";
        document.getElementById("valueInput").focus();
    }
```

```
});
```

The rest of the code in the **charting.js** file is self-explanatory. Let's run the application and see the line chart in action.

# Chapter 13 Running the Real-Time Chart Application

Now for the moment of truth. Build your project and run it.

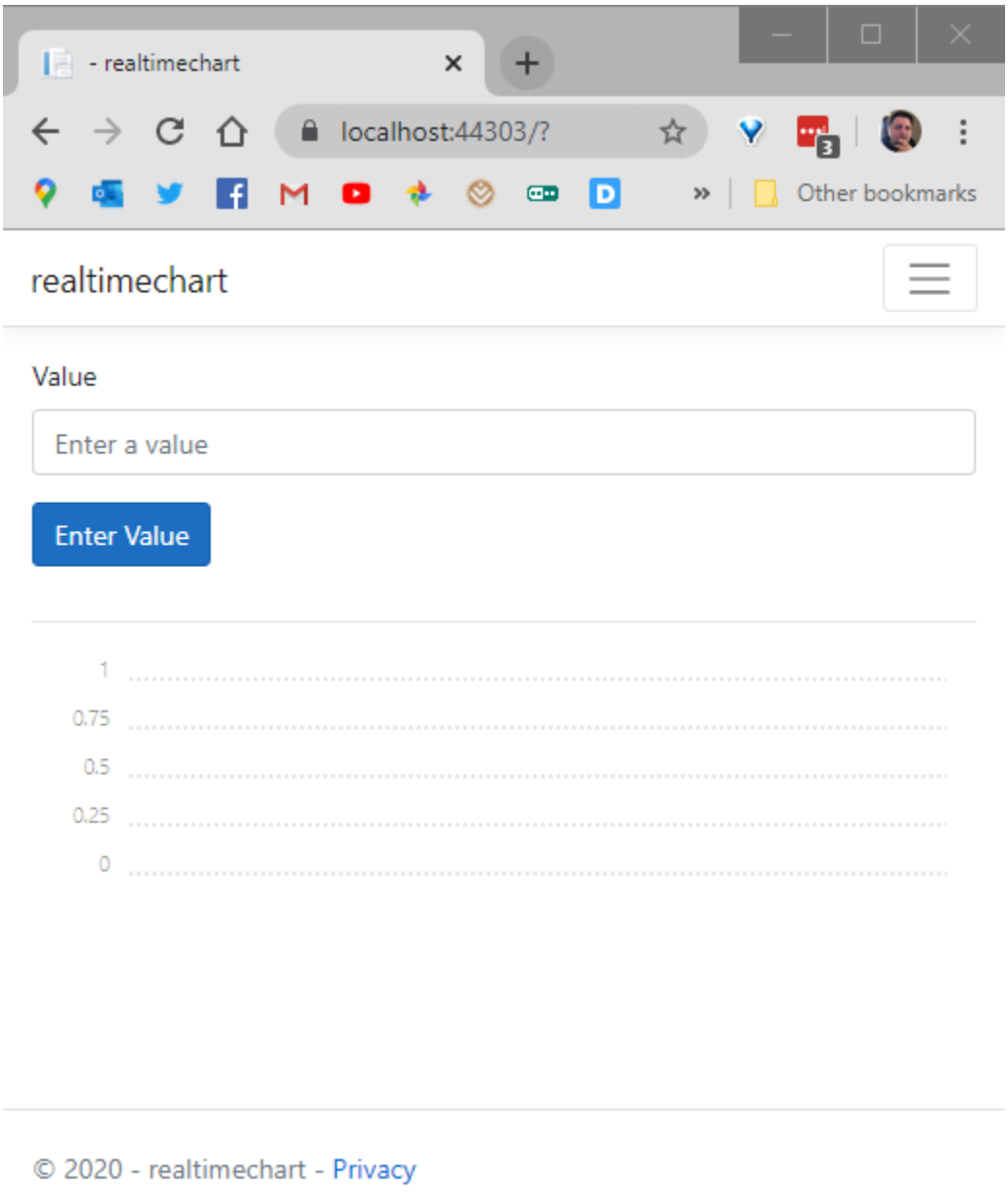


*Figure 39: Running the Chart App*

The application loads the page. If you see the line chart below the button, you know that the page has correctly loaded the **Chartist** CSS file.

Next, add the following values (Code Listing 36) one by one, pressing the **Enter Value** button after each value has been entered.

*Code Listing 36*

```
1, 2, 1.75, 3, 2.2, 1
```

As you enter each value, you will see the line chart render. After adding the last value, your line chart should look like Figure 40.

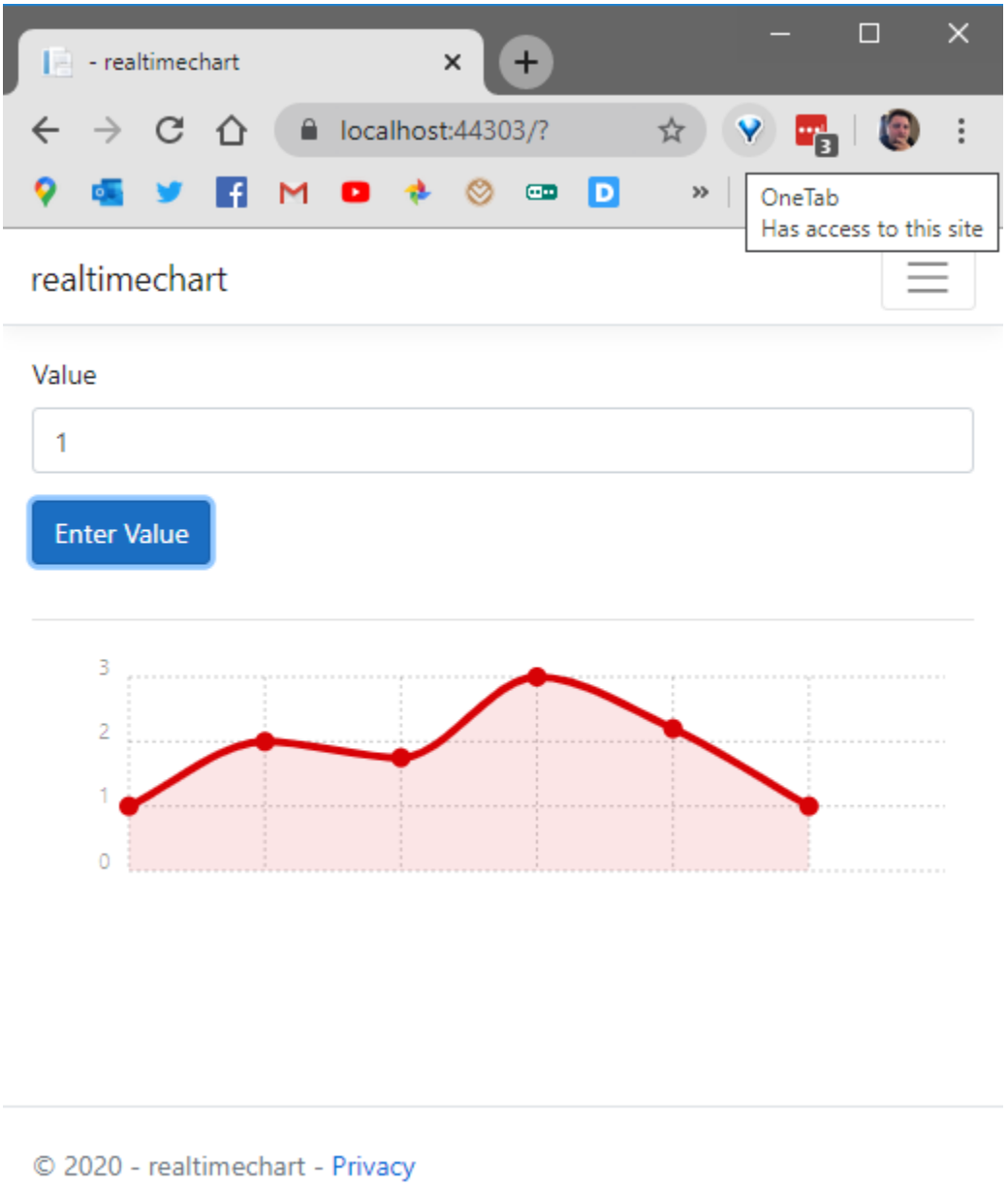


*Figure 40: Data Values Entered*

# Chapter 14 Having Fun with Exchange Rates

Let us have a look at some other fun things to do with SignalR. Let's get an exchange rate chart going. We can leave all the code as is. We are just going to modify our **charting.js** file.

First, we need to find a place from which to grab some real-time exchange rates. For this purpose, Alpha Vantage has a very nice API; to use it, we will need to get a free API key. You can do so by clicking on this link.

> ***Note: Because this is a free API, I will not make more than 5 API requests per minute.***

Head on over to their detailed documentation page and see some of the endpoints that they provide. For this demo, I'll just be using the foreign exchange rate of US dollars (USD) to South African rand (ZAR).

> ***Note: Be sure to add your API key to the end of the URL, after the apikey= parameter.***

For this demo, the endpoint URL looks as follows.

https://www.alphavantage.co/query?function=CURRENCY_EXCHANGE_RATE&from_currency=USD&to_currency=ZAR&apikey=UKHCKPYSU94IQMAQ

Having a look at the JSON returned from the API call, I can see that the exchange rate is looking rather bad (Code Listing 37).

*Code Listing 37*

```
{
    "Realtime Currency Exchange Rate": {
        "1. From_Currency Code": "USD",
        "2. From_Currency Name": "United States Dollar",
        "3. To_Currency Code": "ZAR",
        "4. To_Currency Name": "South African Rand",
        "5. Exchange Rate": "19.03840000",
        "6. Last Refreshed": "2020-04-26 22:02:52",
        "7. Time Zone": "UTC",
        "8. Bid Price": "19.03660000",
        "9. Ask Price": "19.04660000"
    }
}
```

We have a bid price of 19.0366 and an asking price of 19.0466. Modify the code in your **charting.js** file as follows.

Start by adding a timer on an interval of 15 seconds (15,000 milliseconds) that calls a function called **`fireApi`**, as seen in Code Listing 38.

*Code Listing 38*

```
window.setInterval(fireApi, 15000);
```

Next, we need to edit our **`lineChart`** object. We will define the chart as having a low of 19.0 and a high of 19.06. This is based off of the bid price and asking price as seen in the JSON returned from the API call (Code Listing 37).

*Code Listing 39*

```
var lineChart = new Chartist.Line('#chartArea', {
    labels: [],
    series: [[]]
},
{
    low: 19.0,
    high: 19.06,
    showArea: true
});
```

I also want to display the exchange rate somewhere, so I'm just going to repurpose the **`valueInput`** text box for this. Therefore, I need to remove the clearing out of the value and setting the focus after the chart is updated. The commented-out code can be seen in Code Listing 40.

*Code Listing 40*

```
connection.on("ValueReceiver", function (chartValue) {
    if (chartValue && !isNaN(chartValue)) {
        lineChart.data.series[0].push(chartValue);
        lineChart.update();

        //document.getElementById("valueInput").value = "";
        //document.getElementById("valueInput").focus();
    }
});
```

We now need to write an API call in the **`fireApi`** function being called by the timer. Create a function called **`fireApi`** and add the code seen in Code Listing 41. This will call the API, inspect the result, and extract the exchange rate from the JSON.

The exchange rate is parsed as a float and passed to the **ValueSender** method in the hub. The **valueInput** text box is then set to the value of the current exchange rate read from the returned JSON.

*Code Listing 41*

```
function fireApi() {

    $.ajax({
        type: "GET",
        url:
"https://www.alphavantage.co/query?function=CURRENCY_EXCHANGE_RATE&from_cur
rency=USD&to_currency=ZAR&apikey=UKHCKPYSU94IQMAQ",
        dataType: "json",
        success: function (result, status, xhr) {

            var exchRate = result["Realtime Currency Exchange Rate"]["5.
Exchange Rate"];

            if (exchRate) {
                var chartValue = parseFloat(exchRate);

                connection.invoke("ValueSender", chartValue).catch(function
(err) {
                    return console.error(err.toString());
                });
            }

            document.getElementById("valueInput").value = exchRate;
        },
        error: function (xhr, status, error) {
            alert("Result: " + status + " " + error + " " + xhr.status + "
" + xhr.statusText)
        }
    });
}
```

You can see the complete code listing for the modified **charting.js** file in Code Listing 42.

*Code Listing 42*

```
"use strict";

window.setInterval(fireApi, 15000);

var lineChart = new Chartist.Line('#chartArea', {
    labels: [],
    series: [[]]
},
```

```
{
    low: 19.0,
    high: 19.06,
    showArea: true
});

var connection = new
signalR.HubConnectionBuilder().withUrl("/chartHub").build();

//Disable send button until connection is established.
document.getElementById("sendButton").disabled = true;

connection.on("ValueReceiver", function (chartValue) {
    if (chartValue && !isNaN(chartValue)) {
        lineChart.data.series[0].push(chartValue);
        lineChart.update();

        //document.getElementById("valueInput").value = "";
        //document.getElementById("valueInput").focus();
    }
});

connection.start().then(function () {
    document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function
(event) {
    var strValue = document.getElementById("valueInput").value;

    var chartValue = parseFloat(strValue);

    connection.invoke("ValueSender", chartValue).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});

$('#valueInput').keypress(function (e) {
    var key = e.which;
    if (key === 13)  // the enter key code.
    {
        $('#sendButton').click();
        return false;
    }
});
```

```
function fireApi() {

    $.ajax({
        type: "GET",
        url:
"https://www.alphavantage.co/query?function=CURRENCY_EXCHANGE_RATE&from_cur
rency=USD&to_currency=ZAR&apikey=UKHCKPYSU94IQMAQ",
        dataType: "json",
        success: function (result, status, xhr) {

            var exchRate = result["Realtime Currency Exchange Rate"]["5.
Exchange Rate"];

            if (exchRate) {
                var chartValue = parseFloat(exchRate);

                connection.invoke("ValueSender", chartValue).catch(function
(err) {
                    return console.error(err.toString());
                });
            }

            document.getElementById("valueInput").value = exchRate;
        },
        error: function (xhr, status, error) {
            alert("Result: " + status + " " + error + " " + xhr.status + "
" + xhr.statusText)
        }
    });
}
```

Go ahead and run your application, and leave it running for a couple of minutes. Watch as the API is called every 15 seconds, and the exchange rate value is updated in the **Value** text box on the page.

As the value changes after each API call, the line chart is also updated at the same time. After a few minutes, you should end up with a line chart resembling the chart in Figure 41.

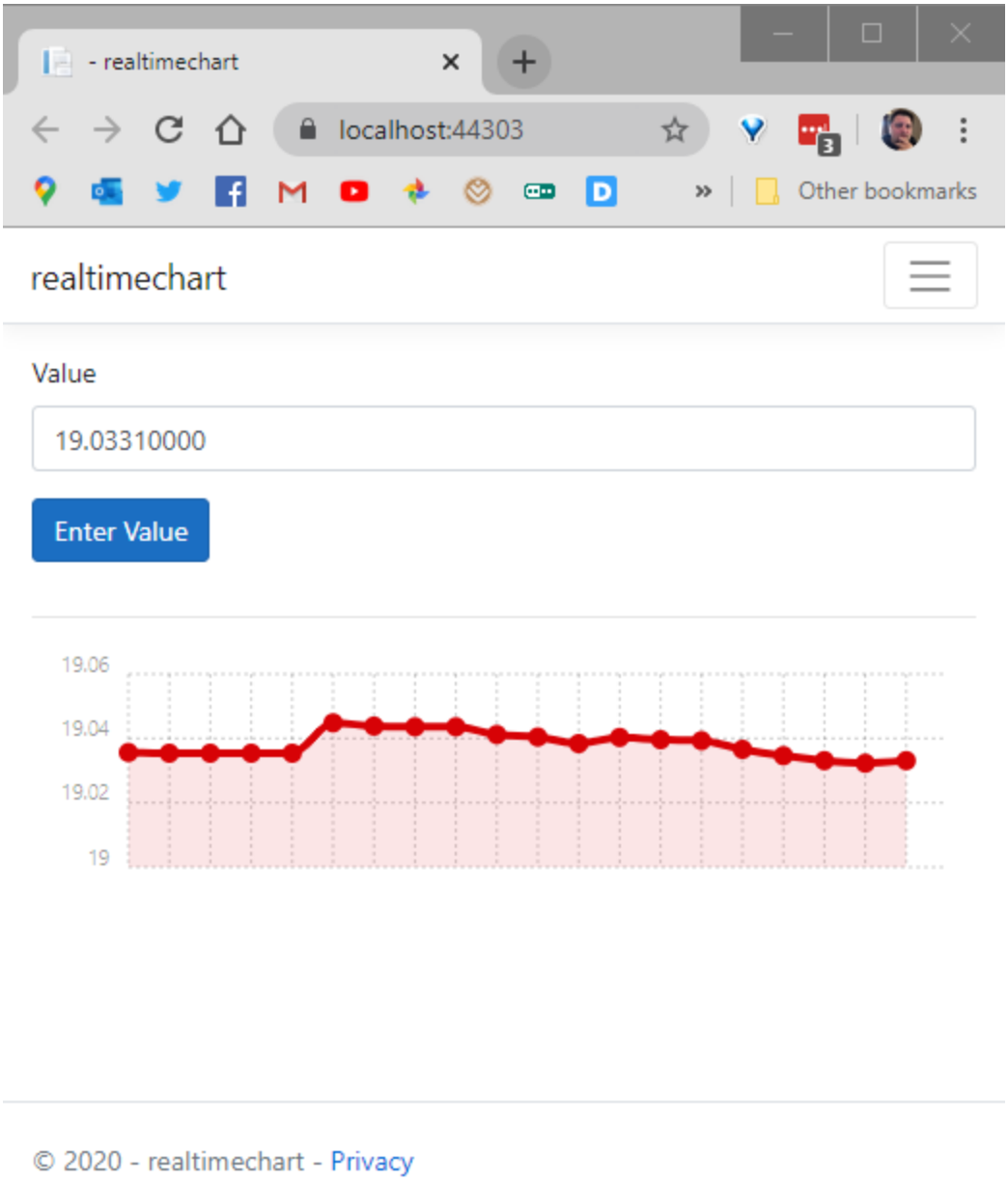


*Figure 41: Exchange Rate Chart*

Play around some more with other API endpoints you might know of. The data can sometimes surprise you once it's charted in a nice line or bar chart.

# Chapter 15 Using Chrome Developer Tools

Chrome developer tools are an absolute lifesaver if you run into any problems. This is true for debugging any web application, especially if you are using scripts or plug-ins. To access Chrome DevTools, right-click anywhere on the page and select **Inspect** from the context menu.

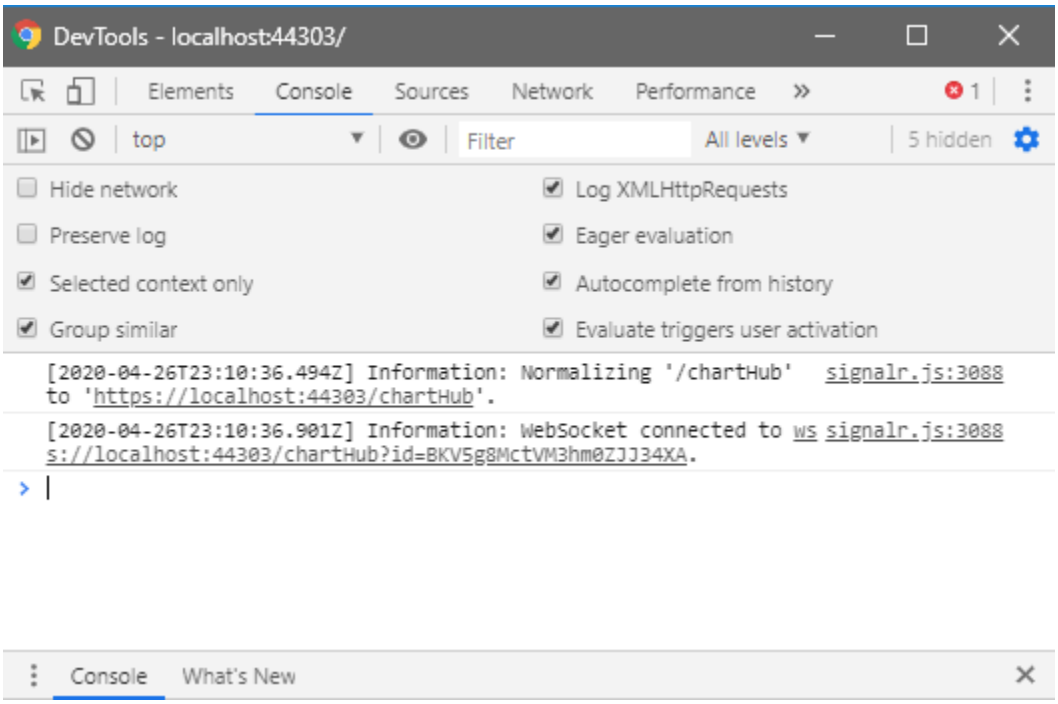


*Figure 42: Console in Chrome DevTools*

Clicking on the **Console** tab (Figure 42), you will see any errors highlighted in red.

***Note: The messages you see in Figure 42 are not error messages. These are normal output messages.***

It is also useful for a quick look at values in your code (without placing a breakpoint). We can achieve this by adding some **`console.log`** statements in the code. Modify the **charting.js** file's **`ValueReceiver`**, as illustrated in Code Listing 43.

*Code Listing 43*

```
connection.on("ValueReceiver", function (chartValue) {
    if (chartValue && !isNaN(chartValue)) {
        lineChart.data.series[0].push(chartValue);
        lineChart.update();

        console.log('Chart value entered was ' + chartValue);
```

```
        document.getElementById("valueInput").value = "";
        document.getElementById("valueInput").focus();

        console.log('Input cleared and Ready');
    }
});
```

Save and refresh your page, enter the value of 1.75 in the **Value** text box, and hit **Enter**. Now, inspect the **Console** tab again.

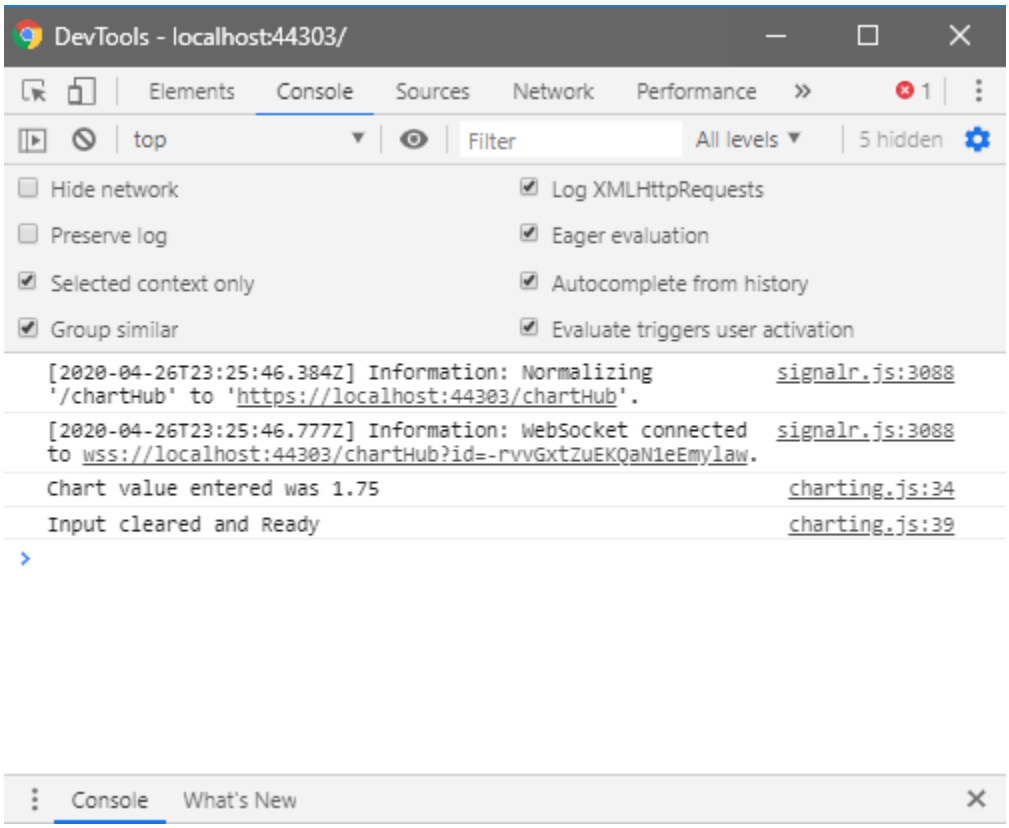


*Figure 43: Viewing Logged Messages*

You will see that the messages you added were logged to the **Console** tab (Figure 43).

You are also able to view **ChartHub** messages. By selecting the **Network** tab, selecting **chartHub** from the left **Name** list, and clicking on the **Messages** sub-tab on the right, you can see the WebSocket messages (Figure 44).

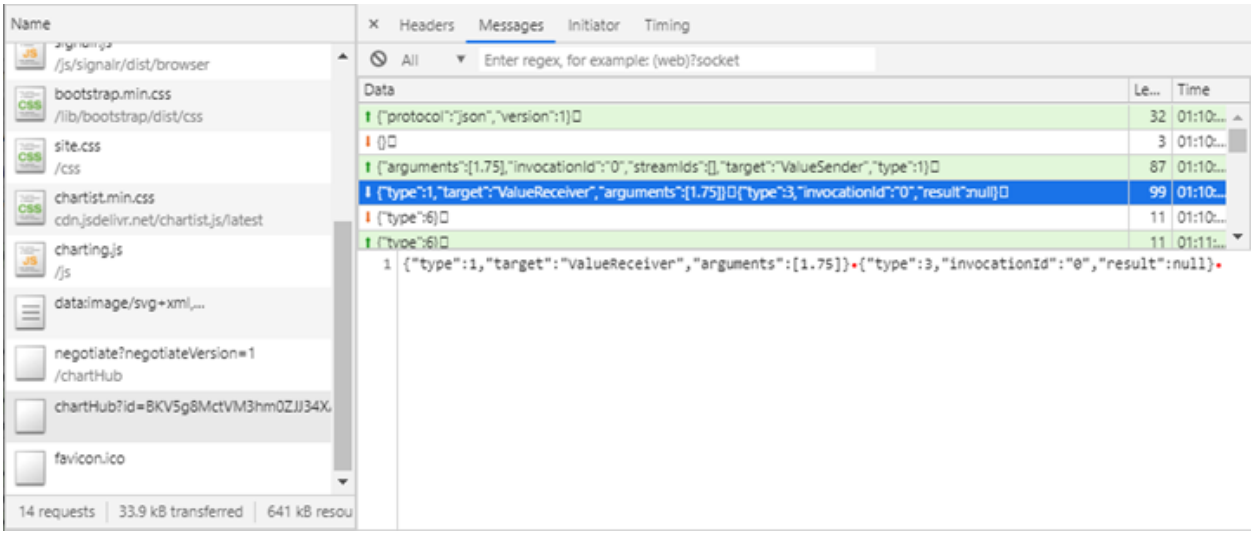


*Figure 44: Viewing ChartHub Messages*

The selected message in Figure 44 can also be seen in Code Listing 44.

*Code Listing 44*

```
{"type":1,"target":"ValueReceiver","arguments":[1.75]}-{"type":3,"invocatio
nId":"0","result":null}
```

Another very helpful tip is the ability to place breakpoints in the JavaScript code. Click the **Sources** tab and expand the folders under the **Page** tree on the left. Look for the **charting.js** file and click on it.

You will see your **charting.js** code displayed as seen in Figure 45. You can then place a breakpoint as I have illustrated, by clicking on line 36. Breakpoints are indicated by a blue tag in the line number gutter.

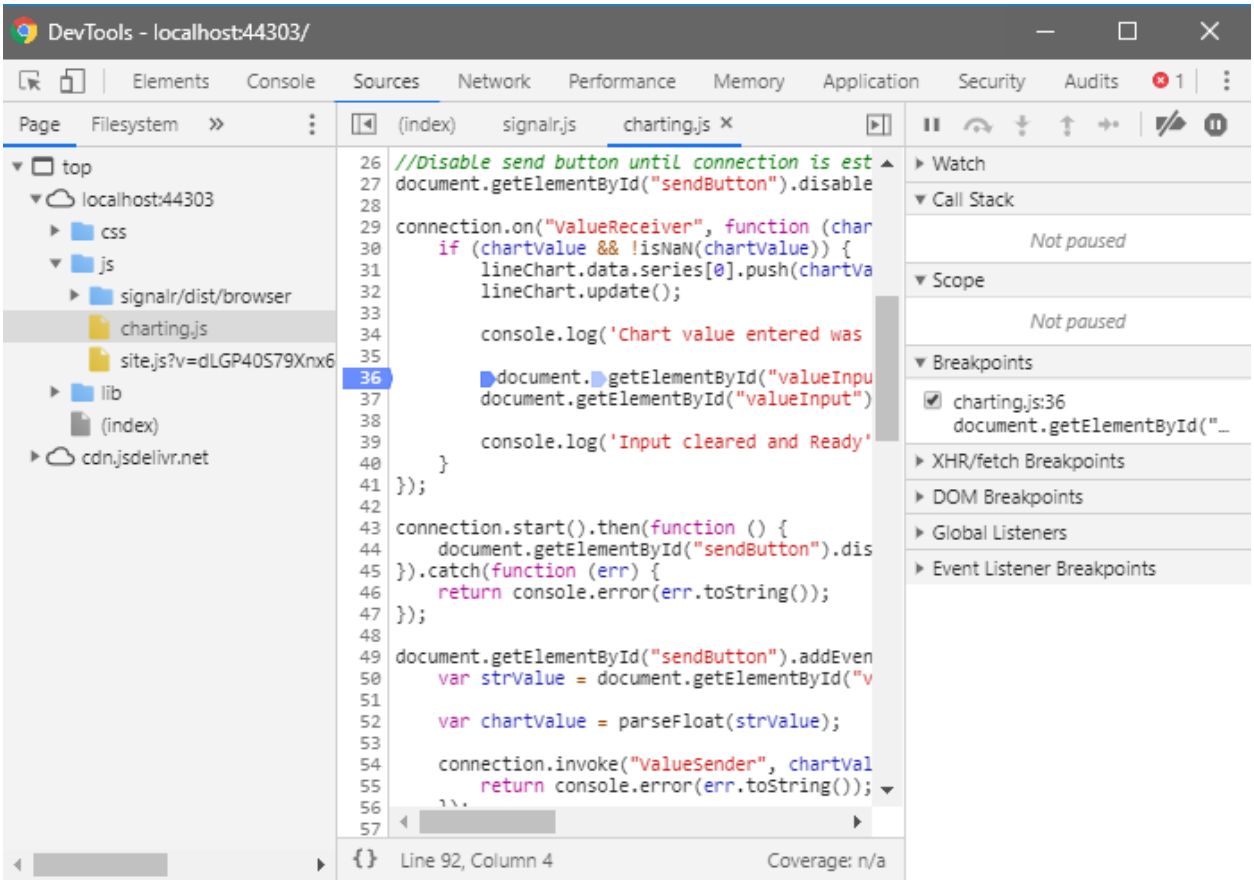


*Figure 45: Setting Breakpoints*

Staying on this screen, you will see that you can set a **Watch**. This allows you to keep track of a variable when it is in scope, and you can see how that value changes.

With your breakpoint still set on line 36, enter a value of **3** into the **Value** text box, as seen in Figure 46, and hit **Enter**. The page will immediately enter a paused state.

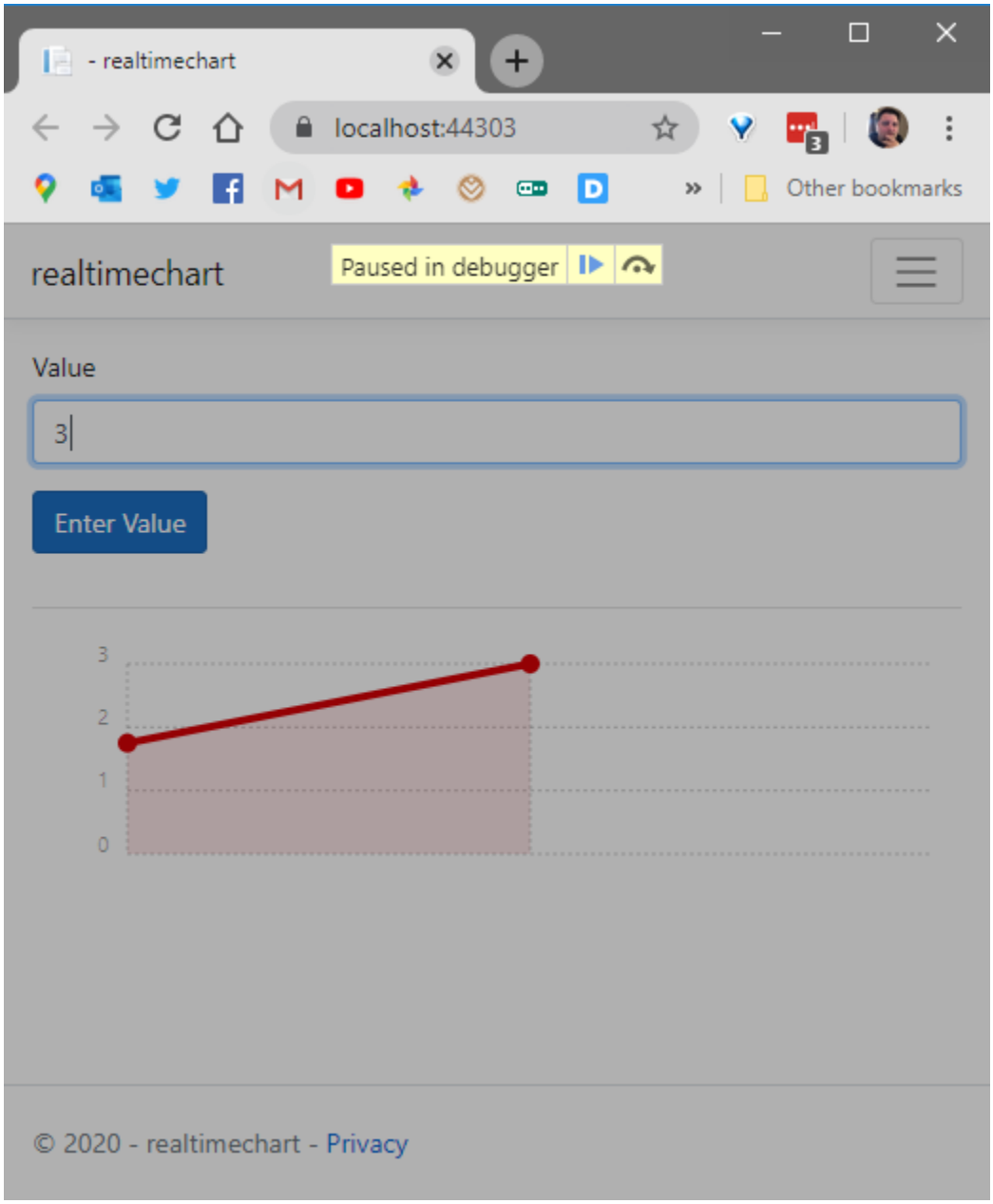


*Figure 46: Enter Value to Hit Breakpoint*

Swing back to the developer tools, and you will see that the execution of your code has paused at line 36 (Figure 47).

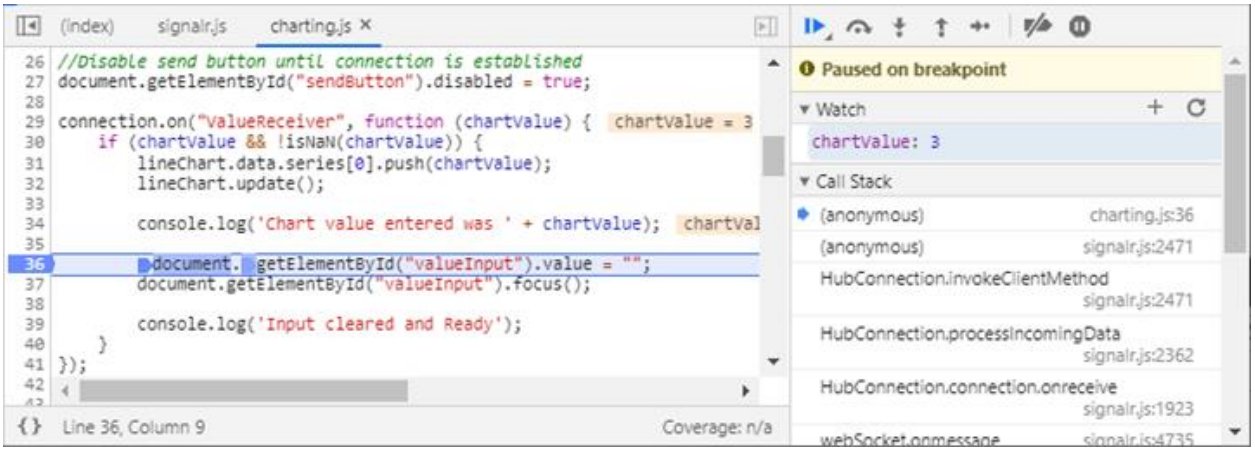


*Figure 47: Add a Watch*

From here, you can expand the **Watch** and add a variable to keep an eye on it. In Figure 47 you will see that I just added the **`chartValue`** variable to a Watch. You will notice that it contains the value that I entered on the page.

# Conclusion

SignalR is very powerful and allows developers to add real-time functionality to their applications. You have seen how to create a chat application with minimal code and how to modify that same code to create a chart application.

By changing a few more lines of code, we were able to automatically pull data via an API call and update our page in real time. Traditionally, this level of functionality was not easy to implement and manage.

There is still much more you can do with Chrome developer tools and ASP.NET Core 3.0 SignalR. Here are some resources you can explore to learn more:

- For more information on ASP.NET Core 3.0 SignalR, have a look at this Microsoft documentation.
- To find out more about what you can do with Chrome DevTools, have a read through Google's detailed documentation.