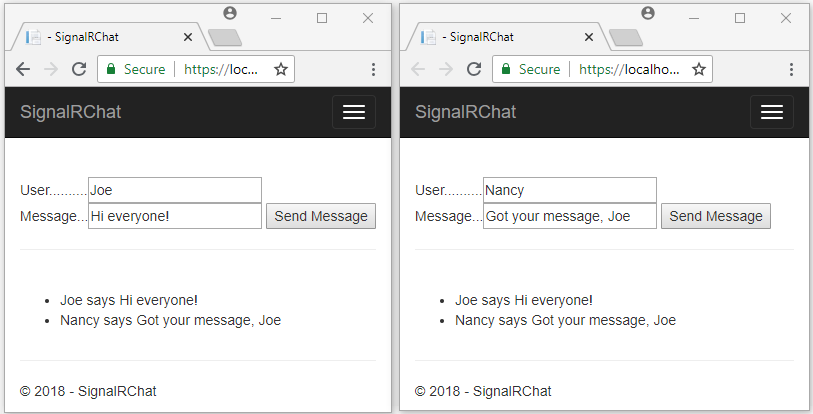
**Tutorial: Get started with ASP.NET Core SignalR**

**Source : ASP.NET Core 2.2 Documentation**

This tutorial teaches the basics of building a real-time app using SignalR. You learn how to:

* Create a web project.
* Add the SignalR client library.
* Create a SignalR hub.
* Configure the project to use SignalR.
* Add code that sends messages from any client to all connected clients.

At the end, you'll have a working chat app:

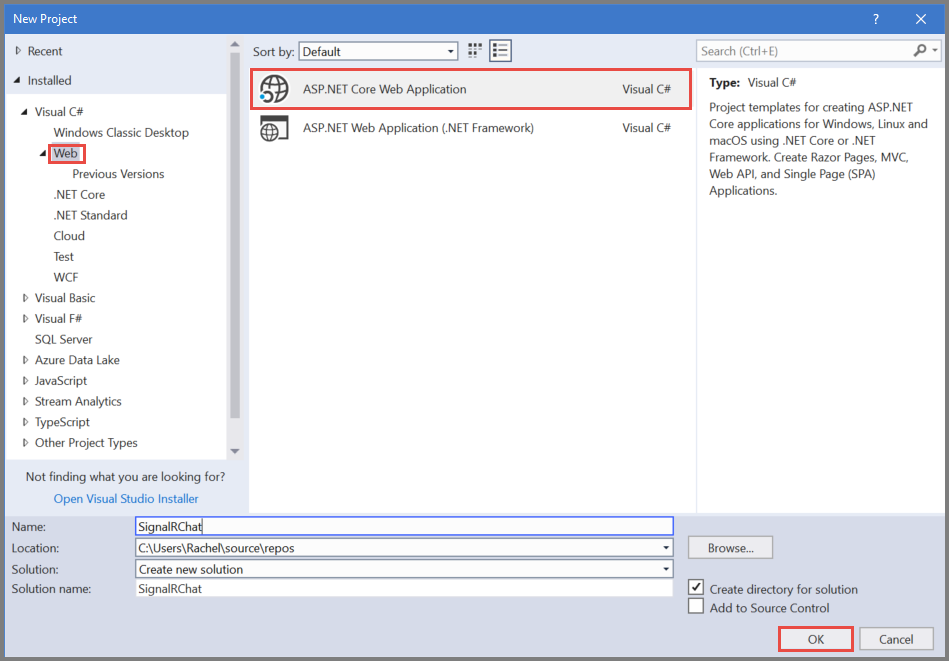


**Prerequisites**

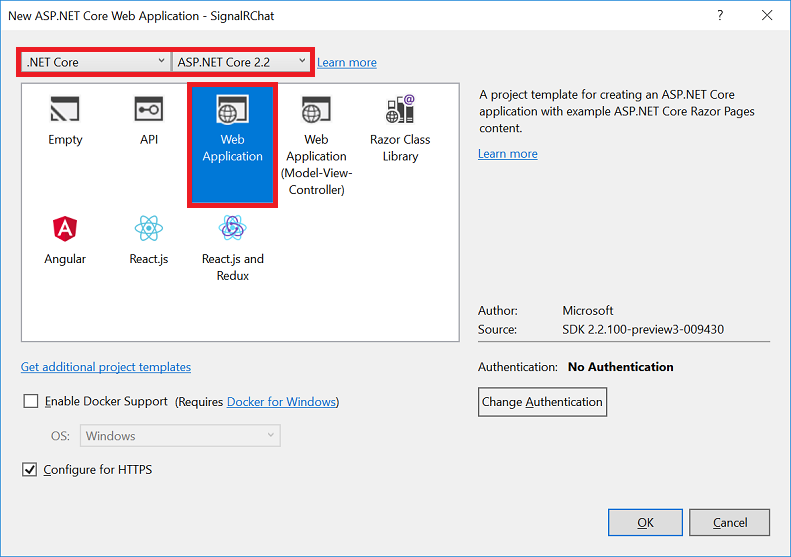
* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-mac)
* [Visual Studio 2017 version 15.9 or later](https://visualstudio.microsoft.com/downloads/) with the **ASP.NET and web development** workload
* [.NET Core SDK 2.2 or later](https://www.microsoft.com/net/download/all)

**Create a web project**

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-mac)
* From the menu, select **File > New Project**.
* In the **New Project** dialog, select **Installed > Visual C# > Web > ASP.NET Core Web Application**. Name the project *SignalRChat*.



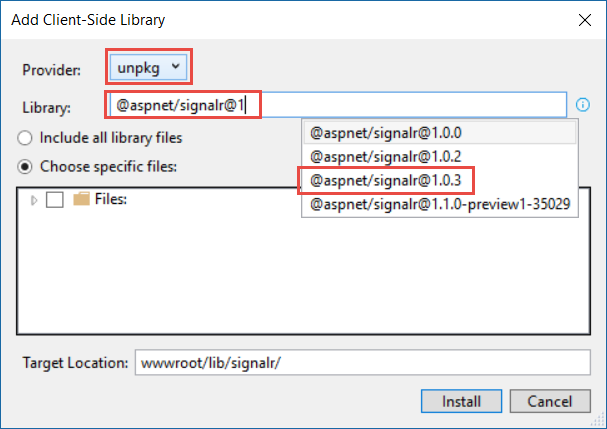
* Select **Web Application** to create a project that uses Razor Pages.
* Select a target framework of **.NET Core**, select **ASP.NET Core 2.2**, and click **OK**.



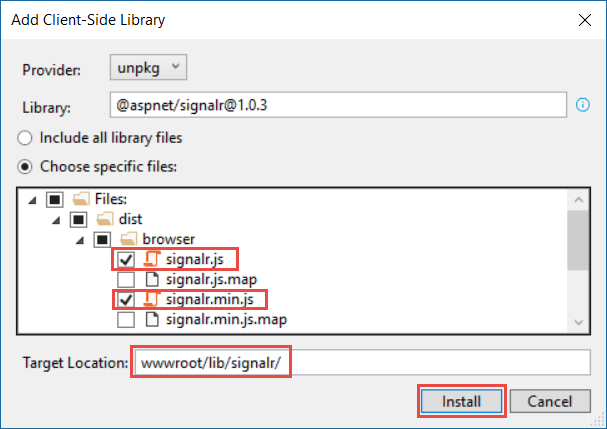
**Add the SignalR client library**

The SignalR server library is included in the Microsoft.AspNetCore.App metapackage. The JavaScript client library isn't automatically included in the project. For this tutorial, you use Library Manager (LibMan) to get the client library from *unpkg*. unpkg is a content delivery network (CDN)) that can deliver anything found in npm, the Node.js package manager.

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio-mac)
* In **Solution Explorer**, right-click the project, and select **Add** > **Client-Side Library**.
* In the **Add Client-Side Library** dialog, for **Provider** select **unpkg**.
* For **Library**, enter @aspnet/signalr@1, and select the latest version that isn't preview.



* Select **Choose specific files**, expand the *dist/browser* folder, and select *signalr.js* and *signalr.min.js*.
* Set **Target Location** to *wwwroot/lib/signalr/*, and select **Install**.



LibMan creates a *wwwroot/lib/signalr* folder and copies the selected files to it.

**Create a SignalR hub**

A *hub* is a class that serves as a high-level pipeline that handles client-server communication.

* In the SignalRChat project folder, create a *Hubs* folder.
* In the *Hubs* folder, create a *ChatHub.cs* file with the following code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace SignarRChatDemo.Hubs

{

using Microsoft.AspNetCore.SignalR;

using System.Threading.Tasks;

public class ChatHub : Hub

{

public async Task SendMessage(string user, string message)

{

await Clients.All.SendAsync("ReceiveMessage", user, message);

}

}

}

* The ChatHub class inherits from the SignalR Hub class. The Hub class manages connections, groups, and messaging.

The SendMessage method can be called by a connected client to send a message to all clients. JavaScript client code that calls the method is shown later in the tutorial. SignalR code is asynchronous to provide maximum scalability.

**Configure SignalR**

The SignalR server must be configured to pass SignalR requests to SignalR.

* Add the following highlighted code to the *Startup.cs* file.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.HttpsPolicy;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using SignarRChatDemo.Hubs;

namespace SignarRChatDemo

{

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

// This method gets called by the runtime. Use this method to add services to the container.

public void ConfigureServices(IServiceCollection services)

{

services.Configure<CookiePolicyOptions>(options =>

{

// This lambda determines whether user consent for non-essential cookies is needed for a given request.

options.CheckConsentNeeded = context => true;

options.MinimumSameSitePolicy = SameSiteMode.None;

});

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

services.AddSignalR();

}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseCookiePolicy();

app.UseSignalR(routes =>

{

routes.MapHub<ChatHub>("/chatHub");

});

app.UseMvc();

}

}

}

* These changes add SignalR to the ASP.NET Core dependency injection system and the middleware pipeline.

**Add SignalR client code**

* Replace the content in *Pages\Index.cshtml* with the following code:

CSHTML

@page

<div class="container">

<div class="row">&nbsp;</div>

<div class="row">

<div class="col-6">&nbsp;</div>

<div class="col-6">

User..........<input type="text" id="userInput" />

<br />

Message...<input type="text" id="messageInput" />

<input type="button" id="sendButton" value="Send Message" />

</div>

</div>

<div class="row">

<div class="col-12">

<hr />

</div>

</div>

<div class="row">

<div class="col-6">&nbsp;</div>

<div class="col-6">

<ul id="messagesList"></ul>

</div>

</div>

</div>

</div>

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

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

The preceding code:

* Creates text boxes for name and message text, and a submit button.
* Creates a list with id="messagesList" for displaying messages that are received from the SignalR hub.
* Includes script references to SignalR and the *chat.js* application code that you create in the next step.

 In the *wwwroot/js* folder, create a *chat.js* file with the following code:

JavaScript

"use strict";

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

//Disable send button until connection is established

document.getElementById("sendButton").disabled = true;

connection.on("ReceiveMessage", function (user, message) {

var msg = message.replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "&gt;");

var encodedMsg = user + " says " + msg;

var li = document.createElement("li");

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("SendMessage", user, message).catch(function (err) {

return console.error(err.toString());

});

event.preventDefault();

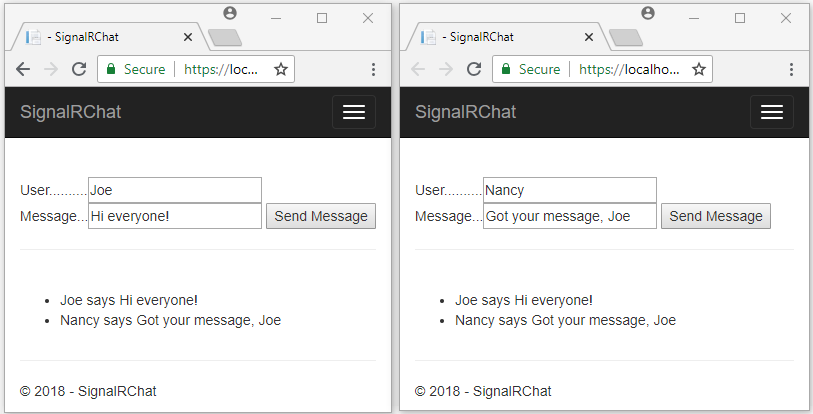
});

* The preceding code:
  + Creates and starts a connection.
  + Adds to the submit button a handler that sends messages to the hub.
  + Adds to the connection object a handler that receives messages from the hub and adds them to the list.

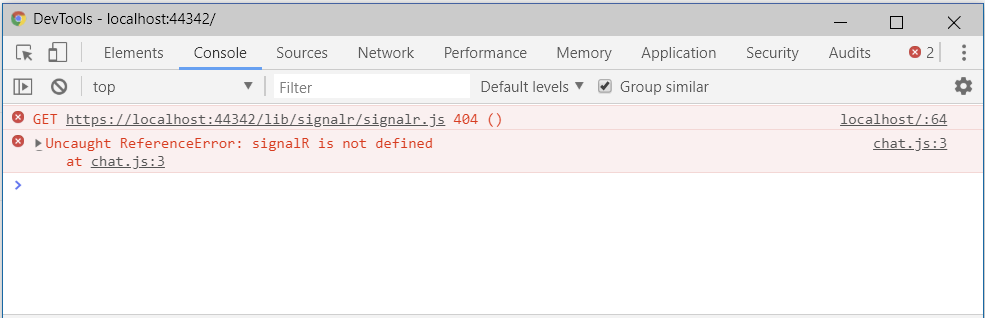
**Run the app**

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-2_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-2_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-2_visual-studio-mac)
* Press **CTRL+F5** to run the app without debugging.
* Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.
* Choose either browser, enter a name and message, and select the **Send Message** button.

The name and message are displayed on both pages instantly.



Tip

If the app doesn't work, open your browser developer tools (F12) and go to the console. You might see errors related to your HTML and JavaScript code. For example, suppose you put *signalr.js* in a different folder than directed. In that case the reference to that file won't work and you'll see a 404 error in the console. 

**Use ASP.NET Core SignalR with TypeScript and Webpack**

[Webpack](https://webpack.js.org/) enables developers to bundle and build the client-side resources of a web app. This tutorial demonstrates using Webpack in an ASP.NET Core SignalR web app whose client is written in [TypeScript](https://www.typescriptlang.org/).

In this tutorial, you learn how to:

* Scaffold a starter ASP.NET Core SignalR app
* Configure the SignalR TypeScript client
* Configure a build pipeline using Webpack
* Configure the SignalR server
* Enable communication between client and server

[View or download sample code](https://github.com/aspnet/Docs/tree/master/aspnetcore/tutorials/signalr-typescript-webpack/sample) ([how to download](https://docs.microsoft.com/en-us/aspnet/core/index?view=aspnetcore-2.2#how-to-download-a-sample))

**Prerequisites**

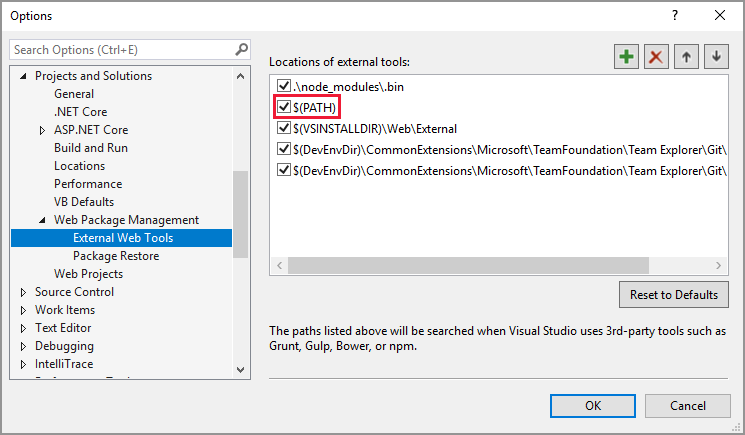
* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr-typescript-webpack?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr-typescript-webpack?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code)
* [Visual Studio 2017 version 15.9 or later](https://visualstudio.microsoft.com/downloads/) with the **ASP.NET and web development** workload
* [.NET Core SDK 2.2 or later](https://www.microsoft.com/net/download/all)

**Create the ASP.NET Core web app**

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr-typescript-webpack?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr-typescript-webpack?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code)

Configure Visual Studio to look for npm in the *PATH* environment variable. By default, Visual Studio uses the version of npm found in its installation directory. Follow these instructions in Visual Studio:

1. Navigate to **Tools** > **Options** > **Projects and Solutions** > **Web Package Management** > **External Web Tools**.
2. Select the *$(PATH)* entry from the list. Click the up arrow to move the entry to the second position in the list.



Visual Studio configuration is completed. It's time to create the project.

1. Use the **File** > **New** > **Project** menu option and choose the **ASP.NET Core Web Application** template.
2. Name the project *SignalRWebPack*, and select **OK**.
3. Select *.NET Core* from the target framework drop-down, and select *ASP.NET Core 2.2* from the framework selector drop-down. Select the **Empty** template, and select **OK**.

**Configure Webpack and TypeScript**

The following steps configure the conversion of TypeScript to JavaScript and the bundling of client-side resources.

1. Execute the following command in the project root to create a *package.json* file:

console

 npm init -y

 Add the highlighted property to the *package.json* file:

JSON

 {

"name": "SignalRWebPack",

"version": "1.0.0",

"private": true,

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "ISC"

}

Setting the private property to true prevents package installation warnings in the next step.

 Install the required npm packages. Execute the following command from the project root:

console

npm install -D -E clean-webpack-plugin@1.0.1 css-loader@2.1.0 html-webpack-plugin@4.0.0-beta.5 mini-css-extract-plugin@0.5.0 ts-loader@5.3.3 typescript@3.3.3 webpack@4.29.3 webpack-cli@3.2.3

Some command details to note:

* A version number follows the @ sign for each package name. npm installs those specific package versions.
* The -E option disables npm's default behavior of writing [semantic versioning](https://semver.org/) range operators to *package.json*. For example, "webpack": "4.29.3" is used instead of "webpack": "^4.29.3". This option prevents unintended upgrades to newer package versions.

See the official [npm-install](https://docs.npmjs.com/cli/install) docs for more detail.

 Replace the scripts property of the *package.json* file with the following snippet:

JSON

 "scripts": {

"build": "webpack --mode=development --watch",

"release": "webpack --mode=production",

"publish": "npm run release && dotnet publish -c Release"

},

Some explanation of the scripts:

* build: Bundles your client-side resources in development mode and watches for file changes. The file watcher causes the bundle to regenerate each time a project file changes. The mode option disables production optimizations, such as tree shaking and minification. Only use build in development.
* release: Bundles your client-side resources in production mode.
* publish: Runs the release script to bundle the client-side resources in production mode. It calls the .NET Core CLI's [publish](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-publish) command to publish the app.

 Create a file named *webpack.config.js*, in the project root, with the following content:

JavaScript

const path = require("path");

const HtmlWebpackPlugin = require("html-webpack-plugin");

const CleanWebpackPlugin = require("clean-webpack-plugin");

const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {

entry: "./src/index.ts",

output: {

path: path.resolve(\_\_dirname, "wwwroot"),

filename: "[name].[chunkhash].js",

publicPath: "/"

},

resolve: {

extensions: [".js", ".ts"]

},

module: {

rules: [

{

test: /\.ts$/,

use: "ts-loader"

},

{

test: /\.css$/,

use: [MiniCssExtractPlugin.loader, "css-loader"]

}

]

},

plugins: [

new CleanWebpackPlugin(["wwwroot/\*"]),

new HtmlWebpackPlugin({

template: "./src/index.html"

}),

new MiniCssExtractPlugin({

filename: "css/[name].[chunkhash].css"

})

]

};

The preceding file configures the Webpack compilation. Some configuration details to note:

* The output property overrides the default value of *dist*. The bundle is instead emitted in the *wwwroot* directory.
* The resolve.extensions array includes *.js* to import the SignalR client JavaScript.

 Create a new *src* directory in the project root. Its purpose is to store the project's client-side assets.

 Create *src/index.html* with the following content.

HTML

 <!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>ASP.NET Core SignalR</title>

</head>

<body>

<div id="divMessages" class="messages">

</div>

<div class="input-zone">

<label id="lblMessage" for="tbMessage">Message:</label>

<input id="tbMessage" class="input-zone-input" type="text" />

<button id="btnSend">Send</button>

</div>

</body>

</html>

The preceding HTML defines the homepage's boilerplate markup.

 Create a new *src/css* directory. Its purpose is to store the project's *.css* files.

 Create *src/css/main.css* with the following content:

css

\*, \*::before, \*::after {

box-sizing: border-box;

}

html, body {

margin: 0;

padding: 0;

}

.input-zone {

align-items: center;

display: flex;

flex-direction: row;

margin: 10px;

}

.input-zone-input {

flex: 1;

margin-right: 10px;

}

.message-author {

font-weight: bold;

}

.messages {

border: 1px solid #000;

margin: 10px;

max-height: 300px;

min-height: 300px;

overflow-y: auto;

padding: 5px;

}

The preceding *main.css* file styles the app.

 Create *src/tsconfig.json* with the following content:

JSON

{

"compilerOptions": {

"target": "es5"

}

}

The preceding code configures the TypeScript compiler to produce [ECMAScript](https://wikipedia.org/wiki/ECMAScript) 5-compatible JavaScript.

 Create *src/index.ts* with the following content:

TypeScript

import "./css/main.css";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");

const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");

const btnSend: HTMLButtonElement = document.querySelector("#btnSend");

const username = new Date().getTime();

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {

if (e.keyCode === 13) {

send();

}

});

btnSend.addEventListener("click", send);

function send() {

}

1. The preceding TypeScript retrieves references to DOM elements and attaches two event handlers:
   * keyup: This event fires when the user types something in the textbox identified as tbMessage. The send function is called when the user presses the **Enter** key.
   * click: This event fires when the user clicks the **Send** button. The send function is called.

**Configure the ASP.NET Core app**

1. The code provided in the Startup.Configure method displays *Hello World!*. Replace the app.Run method call with calls to [UseDefaultFiles](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.defaultfilesextensions.usedefaultfiles#Microsoft_AspNetCore_Builder_DefaultFilesExtensions_UseDefaultFiles_Microsoft_AspNetCore_Builder_IApplicationBuilder_) and [UseStaticFiles](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.staticfileextensions.usestaticfiles#Microsoft_AspNetCore_Builder_StaticFileExtensions_UseStaticFiles_Microsoft_AspNetCore_Builder_IApplicationBuilder_).

C#

app.UseDefaultFiles();

app.UseStaticFiles();

The preceding code allows the server to locate and serve the *index.html* file, whether the user enters its full URL or the root URL of the web app.

 Call [AddSignalR](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.signalrdependencyinjectionextensions.addsignalr#Microsoft_Extensions_DependencyInjection_SignalRDependencyInjectionExtensions_AddSignalR_Microsoft_Extensions_DependencyInjection_IServiceCollection_) in the Startup.ConfigureServices method. It adds the SignalR services to your project.

services.AddSignalR();

 Map a */hub* route to the ChatHub hub. Add the following lines at the end of the Startup.Configure method:

C#

app.UseSignalR(options =>

{

options.MapHub<ChatHub>("/hub");

});

 Create a new directory, called *Hubs*, in the project root. Its purpose is to store the SignalR hub, which is created in the next step.

 Create hub *Hubs/ChatHub.cs* with the following code:

C#

using Microsoft.AspNetCore.SignalR;

using System.Threading.Tasks;

namespace SignalRWebPack.Hubs

{

public class ChatHub : Hub

{

}

}

 Add the following code at the top of the *Startup.cs* file to resolve the ChatHub reference:

using SignalRWebPack.Hubs;

**Enable client and server communication**

The app currently displays a simple form to send messages. Nothing happens when you try to do so. The server is listening to a specific route but does nothing with sent messages.

1. Execute the following command at the project root:

console

npm install @aspnet/signalr

The preceding command installs the [SignalR TypeScript client](https://www.npmjs.com/package/@aspnet/signalr), which allows the client to send messages to the server.

 Add the highlighted code to the *src/index.ts* file:

TypeScript

import "./css/main.css";

import \* as signalR from "@aspnet/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");

const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");

const btnSend: HTMLButtonElement = document.querySelector("#btnSend");

const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()

.withUrl("/hub")

.build();

connection.start().catch(err => document.write(err));

connection.on("messageReceived", (username: string, message: string) => {

let m = document.createElement("div");

m.innerHTML =

`<div class="message-author">${username}</div><div>${message}</div>`;

divMessages.appendChild(m);

divMessages.scrollTop = divMessages.scrollHeight;

});

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {

if (e.keyCode === 13) {

send();

}

});

btnSend.addEventListener("click", send);

function send() {

}

The preceding code supports receiving messages from the server. The HubConnectionBuilder class creates a new builder for configuring the server connection. The withUrl function configures the hub URL.

SignalR enables the exchange of messages between a client and a server. Each message has a specific name. For example, you can have messages with the name messageReceived that execute the logic responsible for displaying the new message in the messages zone. Listening to a specific message can be done via the on function. You can listen to any number of message names. It's also possible to pass parameters to the message, such as the author's name and the content of the message received. Once the client receives a message, a new div element is created with the author's name and the message content in its innerHTML attribute. It's added to the main div element displaying the messages.

 Now that the client can receive a message, configure it to send messages. Add the highlighted code to the *src/index.ts* file:

TypeScript

import "./css/main.css";

import \* as signalR from "@aspnet/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");

const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");

const btnSend: HTMLButtonElement = document.querySelector("#btnSend");

const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()

.withUrl("/hub")

.build();

connection.start().catch(err => document.write(err));

connection.on("messageReceived", (username: string, message: string) => {

let messageContainer = document.createElement("div");

messageContainer.innerHTML =

`<div class="message-author">${username}</div><div>${message}</div>`;

divMessages.appendChild(messageContainer);

divMessages.scrollTop = divMessages.scrollHeight;

});

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {

if (e.keyCode === 13) {

send();

}

});

btnSend.addEventListener("click", send);

function send() {

connection.send("newMessage", username, tbMessage.value)

.then(() => tbMessage.value = "");

}

Sending a message through the WebSockets connection requires calling the send method. The method's first parameter is the message name. The message data inhabits the other parameters. In this example, a message identified as newMessage is sent to the server. The message consists of the username and the user input from a text box. If the send works, the text box value is cleared.

Add the highlighted method to the ChatHub class:

using Microsoft.AspNetCore.SignalR;

using System.Threading.Tasks;

namespace SignalRWebPack.Hubs

{

public class ChatHub : Hub

{

public async Task NewMessage(string username, string message)

{

await Clients.All.SendAsync("messageReceived", username, message);

}

}

}

1. The preceding code broadcasts received messages to all connected users once the server receives them. It's unnecessary to have a generic on method to receive all the messages. A method named after the message name suffices.

In this example, the TypeScript client sends a message identified as newMessage. The C# NewMessage method expects the data sent by the client. A call is made to the [SendAsync](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.signalr.clientproxyextensions.sendasync) method on [Clients.All](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.signalr.ihubclients-1.all). The received messages are sent to all clients connected to the hub.

**Test the app**

Confirm that the app works with the following steps.

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr-typescript-webpack?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr-typescript-webpack?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio-code)

1. Run Webpack in *release* mode. Using the **Package Manager Console** window, execute the following command in the project root. If you are not in the project root, enter cd SignalRWebPack before entering the command.

console

1. npm run release
2. This command yields the client-side assets to be served when running the app. The assets are placed in the *wwwroot* folder.

Webpack completed the following tasks:

* + Purged the contents of the *wwwroot* directory.
  + Converted the TypeScript to JavaScript—a process known as *transpilation*.
  + Mangled the generated JavaScript to reduce file size—a process known as *minification*.
  + Copied the processed JavaScript, CSS, and HTML files from *src* to the *wwwroot* directory.
  + Injected the following elements into the *wwwroot/index.html* file:
    - A <link> tag, referencing the *wwwroot/main.<hash>.css* file. This tag is placed immediately before the closing </head> tag.
    - A <script> tag, referencing the minified *wwwroot/main.<hash>.js* file. This tag is placed immediately before the closing </body> tag.

1. Select **Debug** > **Start without debugging** to launch the app in a browser without attaching the debugger. The *wwwroot/index.html* file is served at http://localhost:<port\_number>.
2. Open another browser instance (any browser). Paste the URL in the address bar.
3. Choose either browser, type something in the **Message** text box, and click the **Send** button. The unique user name and message are displayed on both pages instantly.

