```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSignalR();
```

2. Again, in `Program.cs`, call UseDefaultFiles and UseStaticFiles:

```
C#

var app = builder.Build();

app.UseDefaultFiles();
app.UseStaticFiles();
```

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

3. Create a new directory named `Hubs` in the project root, `SignalRWebpack/`, for the SignalR hub class.

4. Create a new file, `Hubs/ChatHub.cs`, with the following code:

```
C#

using Microsoft.AspNetCore.SignalR;

namespace SignalRWebpack.Hubs;

public class ChatHub : Hub
{
    public async Task NewMessage(long username, string message) =>
        await Clients.All.SendAsync("messageReceived", username,
message);
}
```

   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 is enough.

   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 SendAsync on Clients.All.
   - The received messages are sent to all clients connected to the hub.

5. Add the following `using` statement at the top of `Program.cs` to resolve the `ChatHub` reference:

```C#
using SignalRWebpack.Hubs;
```

6. In `Program.cs`, map the `/hub` route to the `ChatHub` hub. Replace the code that displays `Hello World!` with the following code:

```C#
app.MapHub<ChatHub>("/hub");
```

Configure the client

In this section, you create a Node.js project to convert TypeScript to JavaScript and bundle client-side resources, including HTML and CSS, using Webpack.

1. Run the following command in the project root to create a `package.json` file:

```Console
npm init -y
```

2. Add the highlighted property to the `package.json` file and save the file changes:

```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.

3. Install the required npm packages. Run the following command from the project root:

```
Console

npm i -D -E clean-webpack-plugin css-loader html-webpack-plugin mini-
css-extract-plugin ts-loader typescript webpack webpack-cli
```

The `-E` option disables npm's default behavior of writing semantic versioning range operators to `package.json`. For example, `"webpack": "5.76.1"` is used instead of `"webpack": "^5.76.1"`. This option prevents unintended upgrades to newer package versions.

For more information, see the npm-install documentation.

4. Replace the `scripts` property of `package.json` file with the following code:

```
JSON

"scripts": {
  "build": "webpack --mode=development --watch",
  "release": "webpack --mode=production",
  "publish": "npm run release && dotnet publish -c Release"
},
```

The following scripts are defined:

- `build`: Bundles the 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. use `build` in development only.
- `release`: Bundles the client-side resources in production mode.
- `publish`: Runs the `release` script to bundle the client-side resources in production mode. It calls the .NET CLI's publish command to publish the app.

5. Create a file named `webpack.config.js` in the project root, with the following code:

```
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(),
        new HtmlWebpackPlugin({
            template: "./src/index.html",
        }),
        new MiniCssExtractPlugin({
            filename: "css/[name].[chunkhash].css",
        }),
    ],
};
```

The preceding file configures the Webpack compilation process:

- 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.

6. Create a new directory named `src` in the project root, `SignalRWebpack/`, for the client code.

7. Copy the `src` directory and its contents from the sample project into the project root. The `src` directory contains the following files:

    - `index.html`, which defines the homepage's boilerplate markup:

HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>ASP.NET Core SignalR with TypeScript and
Webpack</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>
```

- `css/main.css`, which provides CSS styles for the homepage:

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;
}
```

- `tsconfig.json`, which configures the TypeScript compiler to produce ECMAScript 5-compatible JavaScript:

JSON

```
{
  "compilerOptions": {
    "target": "es5"
  }
}
```

- `index.ts`:

TypeScript

```
import * as signalR from "@microsoft/signalr";
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();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.on("messageReceived", (username: string, message:
string) => {
  const m = document.createElement("div");

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

  divMessages.appendChild(m);
  divMessages.scrollTop = divMessages.scrollHeight;
});

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

```
tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
  if (e.key === "Enter") {
    send();
  }
});

btnSend.addEventListener("click", send);

function send() {
  connection.send("newMessage", username, tbMessage.value)
    .then(() => (tbMessage.value = ""));
}
```

The preceding code retrieves references to DOM elements and attaches two event handlers:

- `keyup`: Fires when the user types in the `tbMessage` textbox and calls the `send` function when the user presses the **Enter** key.
- `click`: Fires when the user selects the **Send** button and calls `send` function is called.

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, messages with the name `messageReceived` can run the logic responsible for displaying the new message in the messages zone. Listening to a specific message can be done via the `on` function. Any number of message names can be listened to. 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.

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.

8. Run the following command at the project root:

```
Console
```

```
npm i @microsoft/signalr @types/node
```

The preceding command installs:

- The SignalR TypeScript client, which allows the client to send messages to the server.
- The TypeScript type definitions for Node.js, which enables compile-time checking of Node.js types.

Test the app

Confirm that the app works with the following steps:

Visual Studio

1. Run Webpack in `release` mode. Using the **Package Manager Console** window, run the following command in the project root.

   Console

   ```
   npm run release
   ```

   This command generates the client-side assets to be served when running the app. The assets are placed in the `wwwroot` folder.

   Webpack completed the following tasks:

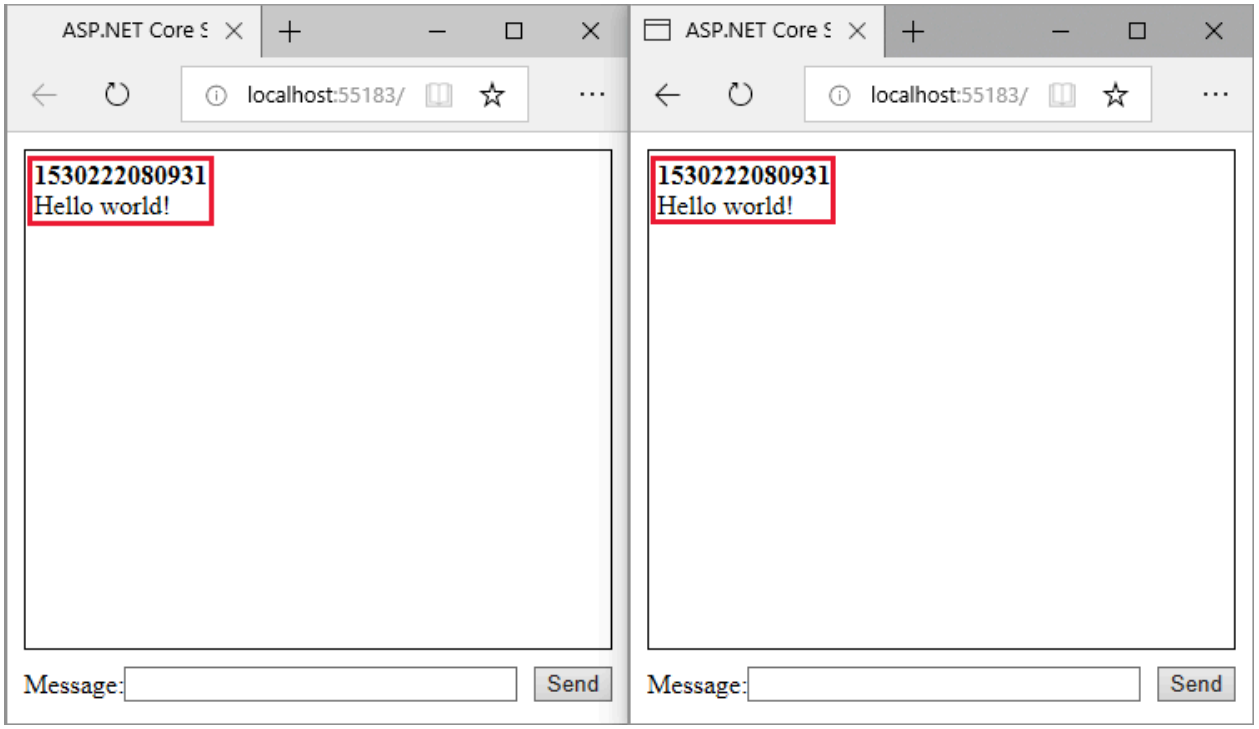
   - Purged the contents of the `wwwroot` directory.
   - Converted the TypeScript to JavaScript in a process known as *transpilation*.
   - Mangled the generated JavaScript to reduce file size in a process known as *minification*.
   - Copied the processed JavaScript, CSS, and HTML files from `src` to 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 after the closing `</title>` tag.

2. Select **Debug** > **Start without debugging** to launch the app in a browser without attaching the debugger. The `wwwroot/index.html` file is served at `https://localhost:<port>`.

   If there are compile errors, try closing and reopening the solution.

3. Open another browser instance (any browser) and paste the URL in the address bar.

4. Choose either browser, type something in the **Message** text box, and select the **Send** button. The unique user name and message are displayed on both pages instantly.



Next steps

- Strongly typed hubs
- Authentication and authorization in ASP.NET Core SignalR
- MessagePack Hub Protocol in SignalR for ASP.NET Core

Additional resources

- ASP.NET Core SignalR JavaScript client
- Use hubs in ASP.NET Core SignalR

Use ASP.NET Core SignalR with Blazor

Article • 06/21/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

This tutorial provides a basic working experience for building a real-time app using SignalR with Blazor. This article is useful for developers who are already familiar with SignalR and are seeking to understand how to use SignalR in a Blazor app. For detailed guidance on the SignalR and Blazor frameworks, see the following reference documentation sets and the API documentation:

- Overview of ASP.NET Core SignalR
- ASP.NET Core Blazor
- .NET API browser

Learn how to:

- ✓ Create a Blazor app
- ✓ Add the SignalR client library
- ✓ Add a SignalR hub
- ✓ Add SignalR services and an endpoint for the SignalR hub
- ✓ Add a Razor component code for chat

At the end of this tutorial, you'll have a working chat app.

Prerequisites

Visual Studio

Visual Studio (latest release) with the **ASP.NET and web development** workload

Sample app

Downloading the tutorial's sample chat app isn't required for this tutorial. The sample app is the final, working app produced by following the steps of this tutorial. When you open the samples repository, open the version folder that you plan to target and find the sample named `BlazorSignalRApp`.

View or download sample code (how to download)

Create a Blazor Web App

Follow the guidance for your choice of tooling:

Visual Studio

> **Note**
>
> Visual Studio 2022 or later and .NET Core SDK 8.0.0 or later are required.

In Visual Studio:

- Select **Create a new project** from the **Start Window** or select **File** > **New** > **Project** from the menu bar.
- In the **Create a new project** dialog, select **Blazor Web App** from the list of project templates. Select the **Next** button.
- In the **Configure your new project** dialog, name the project `BlazorSignalRApp` in the **Project name** field, including matching the capitalization. Using this exact project name is important to ensure that the namespaces match for code that you copy from the tutorial into the app that you're building.
- Confirm that the **Location** for the app is suitable. Leave the **Place solution and project in the same directory** checkbox selected. Select the **Next** button.
- In the **Additional information** dialog, use the following settings:
  - **Framework**: Confirm that the latest framework is selected. If Visual Studio's **Framework** dropdown list doesn't include the latest available .NET framework, update Visual Studio and restart the tutorial.
  - **Authentication type**: **None**
  - **Configure for HTTPS**: Selected
  - **Interactive render mode**: **WebAssembly**
  - **Interactivity location**: **Per page/component**
  - **Include sample pages**: Selected
  - **Do not use top-level statements**: Not selected

- Select **Create**.

The guidance in this article uses a WebAssembly component for the SignalR client because it doesn't make sense to use SignalR to connect to a hub from an Interactive Server component in the same app, as that can lead to server port exhaustion.

Add the SignalR client library

Visual Studio

In **Solution Explorer**, right-click the `BlazorSignalRApp.Client` project and select **Manage NuGet Packages**.

In the **Manage NuGet Packages** dialog, confirm that the **Package source** is set to `nuget.org`.

With **Browse** selected, type `Microsoft.AspNetCore.SignalR.Client` in the search box.

In the search results, select the latest release of the Microsoft.AspNetCore.SignalR.Client package. Select **Install**.

If the **Preview Changes** dialog appears, select **OK**.

If the **License Acceptance** dialog appears, select **I Accept** if you agree with the license terms.

Add a SignalR hub

In the server `BlazorSignalRApp` project, create a `Hubs` (plural) folder and add the following `ChatHub` class (`Hubs/ChatHub.cs`):

C#

```
using Microsoft.AspNetCore.SignalR;

namespace BlazorSignalRApp.Hubs;

public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
```

```
    }
}
```

Add services and an endpoint for the SignalR hub

Open the `Program` file of the server `BlazorSignalRApp` project.

Add the namespaces for Microsoft.AspNetCore.ResponseCompression and the `ChatHub` class to the top of the file:

C#

```
using Microsoft.AspNetCore.ResponseCompression;
using BlazorSignalRApp.Hubs;
```

Add SignalR and Response Compression Middleware services:

C#

```
builder.Services.AddSignalR();

builder.Services.AddResponseCompression(opts =>
{
   opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
       ["application/octet-stream"]);
});
```

Use Response Compression Middleware at the top of the processing pipeline's configuration. Place the following line of code immediately after the line that builds the app (`var app = builder.Build();`):

C#

```
app.UseResponseCompression();
```

Add an endpoint for the hub immediately before the line that runs the app (`app.Run();`):

C#

```
app.MapHub<ChatHub>("/chathub");
```

Add Razor component code for chat

Add the following `Pages/Chat.razor` file to the `BlazorSignalRApp.Client` project:

razor

```razor
@page "/chat"
@rendermode InteractiveWebAssembly
@using Microsoft.AspNetCore.SignalR.Client
@inject NavigationManager Navigation
@implements IAsyncDisposable

<PageTitle>Chat</PageTitle>

<div class="form-group">
    <label>
        User:
        <input @bind="userInput" />
    </label>
</div>
<div class="form-group">
    <label>
        Message:
        <input @bind="messageInput" size="50" />
    </label>
</div>
<button @onclick="Send" disabled="@(!IsConnected)">Send</button>

<hr>

<ul id="messagesList">
    @foreach (var message in messages)
    {
        <li>@message</li>
    }
</ul>

@code {
    private HubConnection? hubConnection;
    private List<string> messages = [];
    private string? userInput;
    private string? messageInput;

    protected override async Task OnInitializedAsync()
    {
        hubConnection = new HubConnectionBuilder()
            .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
            .Build();

        hubConnection.On<string, string>("ReceiveMessage", (user, message)
=>
        {
            var encodedMsg = $"{user}: {message}";
```

```
            messages.Add(encodedMsg);
            InvokeAsync(StateHasChanged);
        });

        await hubConnection.StartAsync();
    }

    private async Task Send()
    {
        if (hubConnection is not null)
        {
            await hubConnection.SendAsync("SendMessage", userInput,
messageInput);
        }
    }

    public bool IsConnected =>
        hubConnection?.State == HubConnectionState.Connected;

    public async ValueTask DisposeAsync()
    {
        if (hubConnection is not null)
        {
            await hubConnection.DisposeAsync();
        }
    }
}
```

Add an entry to the `NavMenu` component to reach the chat page. In `Components/Layout/NavMenu.razor` immediately after the `<div>` block for the `Weather` component, add the following `<div>` block:

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="chat">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Chat
    </NavLink>
</div>
```

> ⓘ Note
>
> Disable Response Compression Middleware in the `Development` environment when using Hot Reload. For more information, see ASP.NET Core Blazor SignalR guidance.

Run the app

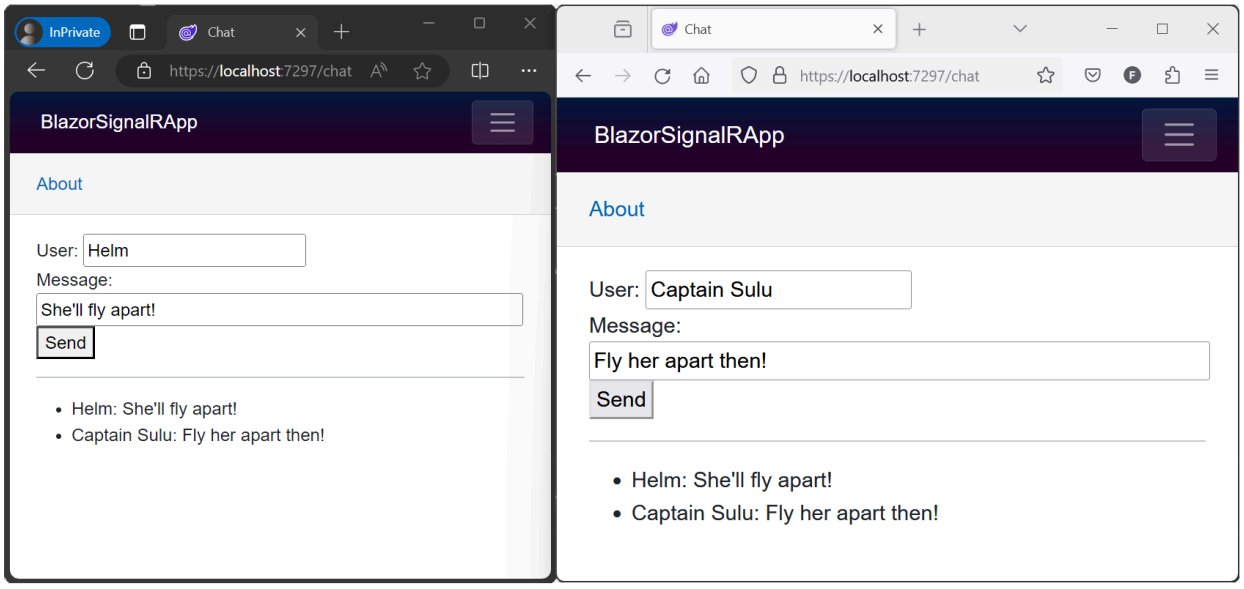
Follow the guidance for your tooling:

Visual Studio

With the server `BlazorSignalRApp` project selected in **Solution Explorer**, press `F5` to run the app with debugging or `Ctrl`+`F5` (Windows)/`⌘`+`F5` (macOS) 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 button to send the message. The name and message are displayed on both pages instantly:



Quotes: *Star Trek VI: The Undiscovered Country* ©1991 Paramount

Next steps

In this tutorial, you learned how to:

- ✓ Create a Blazor app
- ✓ Add the SignalR client library
- ✓ Add a SignalR hub
- ✓ Add SignalR services and an endpoint for the SignalR hub
- ✓ Add a Razor component code for chat

For detailed guidance on the SignalR and Blazor frameworks, see the following reference documentation sets:

Overview of ASP.NET Core SignalR | ASP.NET Core Blazor

Additional resources

- Bearer token authentication with Identity Server, WebSockets, and Server-Sent Events
- Secure a SignalR hub in Blazor WebAssembly apps
- SignalR cross-origin negotiation for authentication
- SignalR configuration
- Debug ASP.NET Core Blazor apps
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

Use hubs in SignalR for ASP.NET Core

Article • 06/18/2024

By Rachel Appel and Kevin Griffin

The SignalR Hubs API enables connected clients to call methods on the server, facilitating real-time communication. The server defines methods that are called by the client, and the client defines methods that are called by the server. SignalR also enables indirect client-to-client communication, always mediated by the SignalR Hub, allowing messages to be sent between individual clients, groups, or to all connected clients. SignalR takes care of everything required to make real-time client-to-server and server-to-client communication possible.

Configure SignalR hubs

To register the services required by SignalR hubs, call AddSignalR in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddSignalR();
```

To configure SignalR endpoints, call MapHub, also in `Program.cs`:

C#

```
app.MapRazorPages();
app.MapHub<ChatHub>("/Chat");

app.Run();
```

> **Note**
>
> ASP.NET Core SignalR server-side assemblies are now installed with the .NET Core SDK. See SignalR assemblies in shared framework for more information.

Create and use hubs

Create a hub by declaring a class that inherits from Hub. Add `public` methods to the class to make them callable from clients:

C#

```
public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
        => await Clients.All.SendAsync("ReceiveMessage", user, message);
}
```

> **Note**
>
> Hubs are transient:
>
> - Don't store state in a property of the hub class. Each hub method call is executed on a new hub instance.
> - Don't instantiate a hub directly via dependency injection. To send messages to a client from elsewhere in your application use an IHubContext.
> - Use `await` when calling asynchronous methods that depend on the hub staying alive. For example, a method such as `Clients.All.SendAsync(...)` can fail if it's called without `await` and the hub method completes before `SendAsync` finishes.

The Context object

The Hub class includes a Context property that contains the following properties with information about the connection:

Expand table

Property	Description
ConnectionId	Gets the unique ID for the connection, assigned by SignalR. There's one connection ID for each connection.
UserIdentifier	Gets the user identifier. By default, SignalR uses the ClaimTypes.NameIdentifier from the ClaimsPrincipal associated with the connection as the user identifier.
User	Gets the ClaimsPrincipal associated with the current user.

Property	Description
Items	Gets a key/value collection that can be used to share data within the scope of this connection. Data can be stored in this collection and it will persist for the connection across different hub method invocations.
Features	Gets the collection of features available on the connection. For now, this collection isn't needed in most scenarios, so it isn't documented in detail yet.
ConnectionAborted	Gets a CancellationToken that notifies when the connection is aborted.

Hub.Context also contains the following methods:

Expand table

Method	Description
GetHttpContext	Returns the HttpContext for the connection, or `null` if the connection isn't associated with an HTTP request. For HTTP connections, use this method to get information such as HTTP headers and query strings.
Abort	Aborts the connection.

The Clients object

The Hub class includes a Clients property that contains the following properties for communication between server and client:

Expand table

Property	Description
All	Calls a method on all connected clients
Caller	Calls a method on the client that invoked the hub method
Others	Calls a method on all connected clients except the client that invoked the method

Hub.Clients also contains the following methods:

Expand table

Method	Description
AllExcept	Calls a method on all connected clients except for the specified connections

Method	Description
Client	Calls a method on a specific connected client
Clients	Calls a method on specific connected clients
Group	Calls a method on all connections in the specified group
GroupExcept	Calls a method on all connections in the specified group, except the specified connections
Groups	Calls a method on multiple groups of connections
OthersInGroup	Calls a method on a group of connections, excluding the client that invoked the hub method
User	Calls a method on all connections associated with a specific user
Users	Calls a method on all connections associated with the specified users

Each property or method in the preceding tables returns an object with a `SendAsync` method. The `SendAsync` method receives the name of the client method to call and any parameters.

The object returned by the `Client` and `Caller` methods also contain an `InvokeAsync` method, which can be used to wait for a result from the client.

Send messages to clients

To make calls to specific clients, use the properties of the `Clients` object. In the following example, there are three hub methods:

- `SendMessage` sends a message to all connected clients, using `Clients.All`.
- `SendMessageToCaller` sends a message back to the caller, using `Clients.Caller`.
- `SendMessageToGroup` sends a message to all clients in the `SignalR Users` group.

C#

```
public async Task SendMessage(string user, string message)
    => await Clients.All.SendAsync("ReceiveMessage", user, message);

public async Task SendMessageToCaller(string user, string message)
    => await Clients.Caller.SendAsync("ReceiveMessage", user, message);

public async Task SendMessageToGroup(string user, string message)
    => await Clients.Group("SignalR Users").SendAsync("ReceiveMessage",
user, message);
```

Strongly typed hubs

A drawback of using `SendAsync` is that it relies on a string to specify the client method to be called. This leaves code open to runtime errors if the method name is misspelled or missing from the client.

An alternative to using `SendAsync` is to strongly type the Hub class with Hub<T>. In the following example, the `ChatHub` client method has been extracted out into an interface called `IChatClient`:

C#

```
public interface IChatClient
{
    Task ReceiveMessage(string user, string message);
}
```

This interface can be used to refactor the preceding `ChatHub` example to be strongly typed:

C#

```
public class StronglyTypedChatHub : Hub<IChatClient>
{
    public async Task SendMessage(string user, string message)
        => await Clients.All.ReceiveMessage(user, message);

    public async Task SendMessageToCaller(string user, string message)
        => await Clients.Caller.ReceiveMessage(user, message);

    public async Task SendMessageToGroup(string user, string message)
        => await Clients.Group("SignalR Users").ReceiveMessage(user,
message);
}
```

Using `Hub<IChatClient>` enables compile-time checking of the client methods. This prevents issues caused by using strings, since `Hub<T>` can only provide access to the methods defined in the interface. Using a strongly typed `Hub<T>` disables the ability to use `SendAsync`.

> **Note**
>
> The `Async` suffix isn't stripped from method names. Unless a client method is defined with `.on('MyMethodAsync')`, don't use `MyMethodAsync` as the name.

Client results

In addition to making calls to clients, the server can request a result from a client. This requires the server to use `ISingleClientProxy.InvokeAsync` and the client to return a result from its `.On` handler.

There are two ways to use the API on the server, the first is to call `Client(...)` or `Caller` on the `Clients` property in a Hub method:

C#

```
public class ChatHub : Hub
{
    public async Task<string> WaitForMessage(string connectionId)
    {
        var message = await Clients.Client(connectionId).InvokeAsync<string>
(
            "GetMessage");
        return message;
    }
}
```

The second way is to call `Client(...)` on an instance of IHubContext<T>:

C#

```
async Task SomeMethod(IHubContext<MyHub> context)
{
    string result = await
context.Clients.Client(connectionID).InvokeAsync<string>(
        "GetMessage");
}
```

Strongly-typed hubs can also return values from interface methods:

C#

```
public interface IClient
{
    Task<string> GetMessage();
}

public class ChatHub : Hub<IClient>
{
    public async Task<string> WaitForMessage(string connectionId)
    {
        string message = await Clients.Client(connectionId).GetMessage();
        return message;
```

```
    }
}
```

Clients return results in their `.On(...)` handlers, as shown below:

.NET client

C#

```
hubConnection.On("GetMessage", async () =>
{
    Console.WriteLine("Enter message:");
    var message = await Console.In.ReadLineAsync();
    return message;
});
```

Typescript client

TypeScript

```
hubConnection.on("GetMessage", async () => {
    let promise = new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("message");
        }, 100);
    });
    return promise;
});
```

Java client

Java

```
hubConnection.onWithResult("GetMessage", () -> {
    return Single.just("message");
});
```

Change the name of a hub method

By default, a server hub method name is the name of the .NET method. To change this default behavior for a specific method, use the HubMethodName attribute. The client should use this name instead of the .NET method name when invoking the method:

C#

```
[HubMethodName("SendMessageToUser")]
public async Task DirectMessage(string user, string message)
    => await Clients.User(user).SendAsync("ReceiveMessage", user, message);
```

Inject services into a hub

Hub constructors can accept services from DI as parameters, which can be stored in properties on the class for use in a hub method.

When injecting multiple services for different hub methods or as an alternative way of writing code, hub methods can also accept services from DI. By default, hub method parameters are inspected and resolved from DI if possible.

C#

```
services.AddSingleton<IDatabaseService, DatabaseServiceImpl>();

// ...

public class ChatHub : Hub
{
    public Task SendMessage(string user, string message, IDatabaseService
dbService)
    {
        var userName = dbService.GetUserName(user);
        return Clients.All.SendAsync("ReceiveMessage", userName, message);
    }
}
```

If implicit resolution of parameters from services isn't desired, disable it with DisableImplicitFromServicesParameters. To explicitly specify which parameters are resolved from DI in hub methods, use the DisableImplicitFromServicesParameters option and use the `[FromServices]` attribute or a custom attribute that implements `IFromServiceMetadata` on the hub method parameters that should be resolved from DI.

C#

```
services.AddSingleton<IDatabaseService, DatabaseServiceImpl>();
services.AddSignalR(options =>
{
    options.DisableImplicitFromServicesParameters = true;
});

// ...
```

```
public class ChatHub : Hub
{
    public Task SendMessage(string user, string message,
        [FromServices] IDatabaseService dbService)
    {
        var userName = dbService.GetUserName(user);
        return Clients.All.SendAsync("ReceiveMessage", userName, message);
    }
}
```

> **Note**
>
> This feature makes use of IServiceProviderIsService, which is optionally implemented by DI implementations. If the app's DI container doesn't support this feature, injecting services into hub methods isn't supported.

Keyed services support in Dependency Injection

Keyed services refers to a mechanism for registering and retrieving Dependency Injection (DI) services using keys. A service is associated with a key by calling AddKeyedSingleton (or `AddKeyedScoped` or `AddKeyedTransient`) to register it. Access a registered service by specifying the key with the [FromKeyedServices] attribute. The following code shows how to use keyed services:

C#

```
using Microsoft.AspNetCore.SignalR;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");

builder.Services.AddRazorPages();
builder.Services.AddSignalR();

var app = builder.Build();

app.MapRazorPages();
app.MapHub<MyHub>("/myHub");

app.Run();

public interface ICache
{
    object Get(string key);
}
public class BigCache : ICache
```

```
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}

public class MyHub : Hub
{
    public void SmallCacheMethod([FromKeyedServices("small")] ICache cache)
    {
        Console.WriteLine(cache.Get("signalr"));
    }

    public void BigCacheMethod([FromKeyedServices("big")] ICache cache)
    {
        Console.WriteLine(cache.Get("signalr"));
    }
}
```

Handle events for a connection

The SignalR Hubs API provides the OnConnectedAsync and OnDisconnectedAsync virtual methods to manage and track connections. Override the `OnConnectedAsync` virtual method to perform actions when a client connects to the hub, such as adding it to a group:

C#

```
public override async Task OnConnectedAsync()
{
    await Groups.AddToGroupAsync(Context.ConnectionId, "SignalR Users");
    await base.OnConnectedAsync();
}
```

Override the `OnDisconnectedAsync` virtual method to perform actions when a client disconnects. If the client disconnects intentionally, such as by calling `connection.stop()`, the `exception` parameter is set to `null`. However, if the client disconnects due to an error, such as a network failure, the `exception` parameter contains an exception that describes the failure:

C#

```
public override async Task OnDisconnectedAsync(Exception? exception)
{
```

```
    await base.OnDisconnectedAsync(exception);
}
```

RemoveFromGroupAsync doesn't need to be called in OnDisconnectedAsync, it's automatically handled for you.

Handle errors

Exceptions thrown in hub methods are sent to the client that invoked the method. On the JavaScript client, the `invoke` method returns a JavaScript Promise. Clients can attach a `catch` handler to the returned promise or use `try`/`catch` with `async`/`await` to handle exceptions:

JavaScript

```
try {
  await connection.invoke("SendMessage", user, message);
} catch (err) {
  console.error(err);
}
```

Connections aren't closed when a hub throws an exception. By default, SignalR returns a generic error message to the client, as shown in the following example:

Output

```
Microsoft.AspNetCore.SignalR.HubException: An unexpected error occurred
invoking 'SendMessage' on the server.
```

Unexpected exceptions often contain sensitive information, such as the name of a database server in an exception triggered when the database connection fails. SignalR doesn't expose these detailed error messages by default as a security measure. For more information on why exception details are suppressed, see Security considerations in ASP.NET Core SignalR.

If an exceptional condition must be propagated to the client, use the HubException class. If a `HubException` is thrown in a hub method, SignalR **sends the entire exception message to the client**, unmodified:

C#

```
public Task ThrowException()
    => throw new HubException("This error will be sent to the client!");
```

> **Note**
>
> SignalR only sends the `Message` property of the exception to the client. The stack trace and other properties on the exception aren't available to the client.

Additional resources

- View or download sample code (how to download)
- Overview of ASP.NET Core SignalR
- ASP.NET Core SignalR JavaScript client
- Publish an ASP.NET Core SignalR app to Azure App Service

Send messages from outside a hub

Article • 06/18/2024

The SignalR hub is the core abstraction for sending messages to clients connected to the SignalR server. It's also possible to send messages from other places in your app using the `IHubContext` service. This article explains how to access a SignalR `IHubContext` to send notifications to clients from outside a hub.

> **Note**
>
> The `IHubContext` is for sending notifications to clients, it is not used to call methods on the `Hub`.

View or download sample code (how to download)

Get an instance of `IHubContext`

In ASP.NET Core SignalR, you can access an instance of `IHubContext` via dependency injection. You can inject an instance of `IHubContext` into a controller, middleware, or other DI service. Use the instance to send messages to clients.

Inject an instance of `IHubContext` in a controller

You can inject an instance of `IHubContext` into a controller by adding it to your constructor:

C#

```
public class HomeController : Controller
{
    private readonly IHubContext<NotificationHub> _hubContext;

    public HomeController(IHubContext<NotificationHub> hubContext)
    {
        _hubContext = hubContext;
    }
}
```

With access to an instance of `IHubContext`, call client methods as if you were in the hub itself:

C#

```
public async Task<IActionResult> Index()
{
    await _hubContext.Clients.All.SendAsync("Notify", $"Home page loaded at:
{DateTime.Now}");
    return View();
}
```

Get an instance of `IHubContext` in middleware

Access the `IHubContext` within the middleware pipeline like so:

C#

```
app.Use(async (context, next) =>
{
    var hubContext = context.RequestServices
                            .GetRequiredService<IHubContext<ChatHub>>();
    //...

    if (next != null)
    {
        await next.Invoke();
    }
});
```

> **Note**
>
> When client methods are called from outside of the `Hub` class, there's no caller associated with the invocation. Therefore, there's no access to the `ConnectionId`, `Caller`, and `Others` properties.
>
> Apps that need to map a user to the connection ID and persist that mapping can do one of the following:
>
> - Persist mapping of single or multiple connections as groups. See Groups in SignalR for more information.
> - Retain connection and user information through a singleton service. See Inject services into a hub for more information. The singleton service can use any storage method, such as:
>   - In-memory storage in a dictionary.
>   - Permanent external storage. For example, a database or Azure Table storage using the Azure.Data.Tables NuGet package.

- Pass the connection ID between clients.

Get an instance of `IHubContext` from IHost

Accessing an `IHubContext` from the web host is useful for integrating with areas outside of ASP.NET Core, for example, using third-party dependency injection frameworks:

C#

```
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();
            var hubContext =
host.Services.GetService(typeof(IHubContext<ChatHub>));
            host.Run();
        }

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

Inject a strongly-typed HubContext

To inject a strongly-typed HubContext, ensure your Hub inherits from `Hub<T>`. Inject it using the `IHubContext<THub, T>` interface rather than `IHubContext<THub>`.

C#

```
public class ChatController : Controller
{
    public IHubContext<ChatHub, IChatClient> _strongChatHubContext { get; }

    public ChatController(IHubContext<ChatHub, IChatClient> chatHubContext)
    {
        _strongChatHubContext = chatHubContext;
    }

    public async Task SendMessage(string user, string message)
    {
        await _strongChatHubContext.Clients.All.ReceiveMessage(user,
message);
```

```
    }
}
```

See Strongly typed hubs for more information.

Use `IHubContext` in generic code

An injected `IHubContext<THub>` instance can be cast to `IHubContext` without a generic `Hub` type specified.

C#

```
class MyHub : Hub
{ }

class MyOtherHub : Hub
{ }

app.Use(async (context, next) =>
{
    var myHubContext = context.RequestServices
                              .GetRequiredService<IHubContext<MyHub>>();
    var myOtherHubContext = context.RequestServices
                              .GetRequiredService<IHubContext<MyOtherHub>>();
    await CommonHubContextMethod((IHubContext)myHubContext);
    await CommonHubContextMethod((IHubContext)myOtherHubContext);

    await next.Invoke();
}

async Task CommonHubContextMethod(IHubContext context)
{
    await context.Clients.All.SendAsync("clientMethod", new Args());
}
```

This is useful when:

- Writing libraries that don't have a reference to the specific `Hub` type the app is using.
- Writing code that is generic and can apply to multiple different `Hub` implementations

Additional resources

- SignalR assemblies in shared framework
- Get started with ASP.NET Core SignalR

- Use hubs in ASP.NET Core SignalR
- Publish an ASP.NET Core SignalR app to Azure App Service

Manage users and groups in SignalR

Article • 06/18/2024

By Brennan Conroy

SignalR allows messages to be sent to all connections associated with a specific user and to named groups of connections.

View or download sample code (how to download)

Users in SignalR

A single user in SignalR can have multiple connections to an app. For example, a user could be connected on their desktop as well as their phone. Each device has a separate SignalR connection, but they're all associated with the same user. If a message is sent to the user, all of the connections associated with that user receive the message. The user identifier for a connection can be accessed by the `Context.UserIdentifier` property in the hub.

By default, SignalR uses the `ClaimTypes.NameIdentifier` from the `ClaimsPrincipal` associated with the connection as the user identifier. To customize this behavior, see Use claims to customize identity handling.

Send a message to a specific user by passing the user identifier to the `User` function in a hub method, as shown in the following example:

> **Note**
>
> The user identifier is case-sensitive.

C#

```
public Task SendPrivateMessage(string user, string message)
{
    return Clients.User(user).SendAsync("ReceiveMessage", message);
}
```

Groups in SignalR

A group is a collection of connections associated with a name. Messages can be sent to all connections in a group. Groups are the recommended way to send to a connection or multiple connections because the groups are managed by the application. A connection can be a member of multiple groups. Groups are ideal for something like a chat application, where each room can be represented as a group.

Add or remove connections from a group

Connections are added to or removed from groups via the `AddToGroupAsync` and `RemoveFromGroupAsync` methods:

C#

```
public async Task AddToGroup(string groupName)
{
    await Groups.AddToGroupAsync(Context.ConnectionId, groupName);

    await Clients.Group(groupName).SendAsync("Send", $"
{Context.ConnectionId} has joined the group {groupName}.");
}

public async Task RemoveFromGroup(string groupName)
{
    await Groups.RemoveFromGroupAsync(Context.ConnectionId, groupName);

    await Clients.Group(groupName).SendAsync("Send", $"
{Context.ConnectionId} has left the group {groupName}.");
}
```

It's safe to add a user to a group multiple times, no exception is thrown in the case that the user already exists in the group.

Group membership isn't preserved when a connection reconnects. The connection needs to rejoin the group when it's re-established. It's not possible to count the members of a group, since this information isn't available if the application is scaled to multiple servers.

Groups are kept in memory, so they won't persist through a server restart. Consider the Azure SignalR service for scenarios requiring group membership to be persisted. For more information, see Azure SignalR

To protect access to resources while using groups, use authentication and authorization functionality in ASP.NET Core. If a user is added to a group only when the credentials are valid for that group, messages sent to that group will only go to authorized users. However, groups are not a security feature. Authentication claims have features that

groups don't, such as expiry and revocation. If a user's permission to access the group is revoked, the app must remove the user from the group explicitly.

> **Note**
>
> Group names are case-sensitive.

Additional resources

- Get started with ASP.NET Core SignalR
- Use hubs in ASP.NET Core SignalR
- Publish an ASP.NET Core SignalR app to Azure App Service

SignalR API design considerations

Article • 06/18/2024

By Andrew Stanton-Nurse

This article provides guidance for building SignalR-based APIs.

Use custom object parameters to ensure backwards-compatibility

Adding parameters to a SignalR hub method (on either the client or the server) is a *breaking change*. This means older clients/servers will get errors when they try to invoke the method without the appropriate number of parameters. However, adding properties to a custom object parameter is **not** a breaking change. This can be used to design compatible APIs that are resilient to changes on the client or the server.

For example, consider a server-side API like the following:

C#

```
public int GetTotalLength(string param1)
{
    return param1.Length;
}
```

The JavaScript client calls this method using `invoke` as follows:

TypeScript

```
connection.invoke("GetTotalLength", "value1");
```

If you later add a second parameter to the server method, older clients won't provide this parameter value. For example:

C#

```
public int GetTotalLength(string param1, string param2)
{
    return param1.Length + param2.Length;
}
```

When the old client tries to invoke this method, it will get an error like this:

```
Microsoft.AspNetCore.SignalR.HubException: Failed to invoke 'GetTotalLength'
due to an error on the server.
```

On the server, you'll see a log message like this:

```
System.IO.InvalidDataException: Invocation provides 1 argument(s) but target
expects 2.
```

The old client only sent one parameter, but the newer server API required two parameters. Using custom objects as parameters gives you more flexibility. Let's redesign the original API to use a custom object:

C#

```
public class TotalLengthRequest
{
    public string Param1 { get; set; }
}

public int GetTotalLength(TotalLengthRequest req)
{
    return req.Param1.Length;
}
```

Now, the client uses an object to call the method:

TypeScript

```
connection.invoke("GetTotalLength", { param1: "value1" });
```

Instead of adding a parameter, add a property to the `TotalLengthRequest` object:

C#

```
public class TotalLengthRequest
{
    public string Param1 { get; set; }
    public string Param2 { get; set; }
}

public int GetTotalLength(TotalLengthRequest req)
{
```

```
    var length = req.Param1.Length;
    if (req.Param2 != null)
    {
        length += req.Param2.Length;
    }
    return length;
}
```

When the old client sends a single parameter, the extra `Param2` property will be left `null`. You can detect a message sent by an older client by checking the `Param2` for `null` and apply a default value. A new client can send both parameters.

TypeScript

```
connection.invoke("GetTotalLength", { param1: "value1", param2: "value2" });
```

The same technique works for methods defined on the client. You can send a custom object from the server side:

C#

```
public async Task Broadcast(string message)
{
    await Clients.All.SendAsync("ReceiveMessage", new
    {
        Message = message
    });
}
```

On the client side, you access the `Message` property rather than using a parameter:

TypeScript

```
connection.on("ReceiveMessage", (req) => {
    appendMessageToChatWindow(req.message);
});
```

If you later decide to add the sender of the message to the payload, add a property to the object:

C#

```
public async Task Broadcast(string message)
{
    await Clients.All.SendAsync("ReceiveMessage", new
    {
        Sender = Context.User.Identity.Name,
```

```
        Message = message
    });
}
```

The older clients won't be expecting the `Sender` value, so they'll ignore it. A new client can accept it by updating to read the new property:

TypeScript

```
connection.on("ReceiveMessage", (req) => {
    let message = req.message;
    if (req.sender) {
        message = req.sender + ": " + message;
    }
    appendMessageToChatWindow(message);
});
```

In this case, the new client is also tolerant of an old server that doesn't provide the `Sender` value. Since the old server won't provide the `Sender` value, the client checks to see if it exists before accessing it.

Additional resources

- SignalR assemblies in shared framework

Use hub filters in ASP.NET Core SignalR

Article • 06/18/2024

Hub filters:

- Are available in ASP.NET Core 5.0 or later.
- Allow logic to run before and after hub methods are invoked by clients.

This article provides guidance for writing and using hub filters.

Configure hub filters

Hub filters can be applied globally or per hub type. The order in which filters are added is the order in which the filters run. Global hub filters run before local hub filters.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(options =>
    {
        // Global filters will run first
        options.AddFilter<CustomFilter>();
    }).AddHubOptions<ChatHub>(options =>
    {
        // Local filters will run second
        options.AddFilter<CustomFilter2>();
    });
}
```

A hub filter can be added in one of the following ways:

- Add a filter by concrete type:

  C#

  ```
  hubOptions.AddFilter<TFilter>();
  ```

  This will be resolved from dependency injection (DI) or type activated.

- Add a filter by runtime type:

  C#

```
hubOptions.AddFilter(typeof(TFilter));
```

This will be resolved from DI or type activated.

- Add a filter by instance:

```
C#
hubOptions.AddFilter(new MyFilter());
```

This instance will be used like a singleton. All hub method invocations will use the same instance.

Hub filters are created and disposed per hub invocation. If you want to store global state in the filter, or no state, add the hub filter type to DI as a singleton for better performance. Alternatively, add the filter as an instance if you can.

Create hub filters

Create a filter by declaring a class that inherits from `IHubFilter`, and add the `InvokeMethodAsync` method. There is also `OnConnectedAsync` and `OnDisconnectedAsync` that can optionally be implemented to wrap the `OnConnectedAsync` and `OnDisconnectedAsync` hub methods respectively.

```
C#
public class CustomFilter : IHubFilter
{
    public async ValueTask<object> InvokeMethodAsync(
        HubInvocationContext invocationContext, Func<HubInvocationContext,
ValueTask<object>> next)
    {
        Console.WriteLine($"Calling hub method
'{invocationContext.HubMethodName}'");
        try
        {
            return await next(invocationContext);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Exception calling
'{invocationContext.HubMethodName}': {ex}");
            throw;
        }
    }
```

```
    // Optional method
    public Task OnConnectedAsync(HubLifetimeContext context,
Func<HubLifetimeContext, Task> next)
    {
        return next(context);
    }

    // Optional method
    public Task OnDisconnectedAsync(
        HubLifetimeContext context, Exception exception,
Func<HubLifetimeContext, Exception, Task> next)
    {
        return next(context, exception);
    }
}
```

Filters are very similar to middleware. The `next` method invokes the next filter. The final filter will invoke the hub method. Filters can also store the result from awaiting `next` and run logic after the hub method has been called before returning the result from `next`.

To skip a hub method invocation in a filter, throw an exception of type `HubException` instead of calling `next`. The client will receive an error if it was expecting a result.

Use hub filters

When writing the filter logic, try to make it generic by using attributes on hub methods instead of checking for hub method names.

Consider a filter that will check a hub method argument for banned phrases and replace any phrases it finds with `***`. For this example, assume a `LanguageFilterAttribute` class is defined. The class has a property named `FilterArgument` that can be set when using the attribute.

1. Place the attribute on the hub method that has a string argument to be cleaned:

   C#

   ```
   public class ChatHub
   {
       [LanguageFilter(filterArgument = 0)]
       public async Task SendMessage(string message, string username)
       {
           await Clients.All.SendAsync("SendMessage", $"{username} says:
   {message}");
       }
   }
   ```

2. Define a hub filter to check for the attribute and replace banned phrases in a hub method argument with `***`:

```
C#
```

```
public class LanguageFilter : IHubFilter
{
    // populated from a file or inline
    private List<string> bannedPhrases = new List<string> { "async
void", ".Result" };

    public async ValueTask<object>
InvokeMethodAsync(HubInvocationContext invocationContext,
        Func<HubInvocationContext, ValueTask<object>> next)
    {
        var languageFilter =
(LanguageFilterAttribute)Attribute.GetCustomAttribute(
            invocationContext.HubMethod,
typeof(LanguageFilterAttribute));
        if (languageFilter != null &&
            invocationContext.HubMethodArguments.Count >
languageFilter.FilterArgument &&

invocationContext.HubMethodArguments[languageFilter.FilterArgument] is
string str)
        {
            foreach (var bannedPhrase in bannedPhrases)
            {
                str = str.Replace(bannedPhrase, "***");
            }

            var arguments =
invocationContext.HubMethodArguments.ToArray();
            arguments[languageFilter.FilterArgument] = str;
            invocationContext = new
HubInvocationContext(invocationContext.Context,
                invocationContext.ServiceProvider,
                invocationContext.Hub,
                invocationContext.HubMethod,
                arguments);
        }

        return await next(invocationContext);
    }
}
```

3. Register the hub filter in the `Startup.ConfigureServices` method. To avoid reinitializing the banned phrases list for every invocation, the hub filter is registered as a singleton:

```
C#
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(hubOptions =>
    {
        hubOptions.AddFilter<LanguageFilter>();
    });

    services.AddSingleton<LanguageFilter>();
}
```

The HubInvocationContext object

The `HubInvocationContext` contains information for the current hub method invocation.

Expand table

Property	Description	Type
`Context`	The `HubCallerContext` contains information about the connection.	`HubCallerContext`
`Hub`	The instance of the Hub being used for this hub method invocation.	`Hub`
`HubMethodName`	The name of the hub method being invoked.	`string`
`HubMethodArguments`	The list of arguments being passed to the hub method.	`IReadOnlyList<string>`
`ServiceProvider`	The scoped service provider for this hub method invocation.	`IServiceProvider`
`HubMethod`	The hub method information.	`MethodInfo`

The HubLifetimeContext object

The `HubLifetimeContext` contains information for the `OnConnectedAsync` and `OnDisconnectedAsync` hub methods.

Expand table

Property	Description	Type
`Context`	The `HubCallerContext` contains information about the connection.	`HubCallerContext`

Property	Description	Type
`Hub`	The instance of the Hub being used for this hub method invocation.	`Hub`
`ServiceProvider`	The scoped service provider for this hub method invocation.	`IServiceProvider`

Authorization and filters

Authorize attributes on hub methods run before hub filters.

ASP.NET Core SignalR clients

Article • 06/18/2024

Versioning, support, and compatibility

The SignalR clients ship alongside the server components and are versioned to match. Any supported client can safely connect to any supported server, and any compatibility issues would be considered bugs to be fixed. SignalR clients are supported in the same support lifecycle as the rest of .NET Core. See the .NET Core Support Policy for details.

Many features require a compatible client **and** server. See below for a table showing the minimum versions for various features.

The 1.x versions of SignalR map to the 2.1 and 2.2 .NET Core releases and have the same lifetime. For version 3.x and above, the SignalR version exactly matches the rest of .NET and has the same support lifecycle.

Expand table

SignalR version	.NET Core version	Support level	End of support
1.0.x	2.1.x	Long Term Support	August 21, 2021
1.1.x	2.2.x	End Of Life	December 23, 2019
3.x or higher	*same as SignalR version*	See the the .NET Core Support Policy	

NOTE: In ASP.NET Core 3.0, the JavaScript client *moved* to the `@microsoft/signalr` npm package.

Feature distribution

The table below shows the features and support for the clients that offer real-time support. For each feature, the *minimum* version supporting this feature is listed. If no version is listed, the feature isn't supported.

Expand table

Feature	Server	.NET client	JavaScript client	Java client
Azure SignalR Service Support	2.1.0	1.0.0	1.0.0	1.0.0
Server-to-client Streaming	2.1.0	1.0.0	1.0.0	1.0.0
Client-to-server Streaming	3.0.0	3.0.0	3.0.0	3.0.0
Automatic Reconnection (.NET, JavaScript)	3.0.0	3.0.0	3.0.0	❌
WebSockets Transport	2.1.0	1.0.0	1.0.0	1.0.0
Server-Sent Events Transport	2.1.0	1.0.0	1.0.0	❌
Long Polling Transport	2.1.0	1.0.0	1.0.0	3.0.0
JSON Hub Protocol	2.1.0	1.0.0	1.0.0	1.0.0
MessagePack Hub Protocol	2.1.0	1.0.0	1.0.0	5.0.0
Client Results	7.0.0	7.0.0	7.0.0	7.0.0

Support for enabling additional client features is tracked in our issue tracker.

Browsers that don't support ECMAScript 6 (ES6)

SignalR targets ES6. For browsers that don't support ES6, transpile the library to ES5. For more information, see Getting Started with ES6 – Transpiling ES6 to ES5 with Traceur and Babel.

Additional resources

- Get started with SignalR for ASP.NET Core
- Supported platforms
- Hubs
- JavaScript client

ASP.NET Core SignalR .NET Client

Article • 06/18/2024

The ASP.NET Core SignalR .NET client library lets you communicate with SignalR hubs from .NET apps.

View or download sample code (how to download)

The code sample in this article is a WPF app that uses the ASP.NET Core SignalR .NET client.

Install the SignalR .NET client package

The Microsoft.AspNetCore.SignalR.Client package is required for .NET clients to connect to SignalR hubs.

Visual Studio

To install the client library, run the following command in the **Package Manager Console** window:

PowerShell

```
Install-Package Microsoft.AspNetCore.SignalR.Client
```

Connect to a hub

To establish a connection, create a `HubConnectionBuilder` and call `Build`. The hub URL, protocol, transport type, log level, headers, and other options can be configured while building a connection. Configure any required options by inserting any of the `HubConnectionBuilder` methods into `Build`. Start the connection with `StartAsync`.

C#

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

namespace SignalRChatClient
{
    public partial class MainWindow : Window
```

```
    {
        HubConnection connection;
        public MainWindow()
        {
            InitializeComponent();

            connection = new HubConnectionBuilder()
                .WithUrl("http://localhost:53353/ChatHub")
                .Build();

            connection.Closed += async (error) =>
            {
                await Task.Delay(new Random().Next(0,5) * 1000);
                await connection.StartAsync();
            };
        }

        private async void connectButton_Click(object sender,
RoutedEventArgs e)
        {
            connection.On<string, string>("ReceiveMessage", (user, message)
=>
            {
                this.Dispatcher.Invoke(() =>
                {
                   var newMessage = $"{user}: {message}";
                   messagesList.Items.Add(newMessage);
                });
            });

            try
            {
                await connection.StartAsync();
                messagesList.Items.Add("Connection started");
                connectButton.IsEnabled = false;
                sendButton.IsEnabled = true;
            }
            catch (Exception ex)
            {
                messagesList.Items.Add(ex.Message);
            }
        }

        private async void sendButton_Click(object sender, RoutedEventArgs
e)
        {
            try
            {
                await connection.InvokeAsync("SendMessage",
                    userTextBox.Text, messageTextBox.Text);
            }
            catch (Exception ex)
            {
                messagesList.Items.Add(ex.Message);
            }
```

```
        }
    }
}
```

Handle lost connection

Automatically reconnect

The HubConnection can be configured to automatically reconnect using the `WithAutomaticReconnect` method on the HubConnectionBuilder. It won't automatically reconnect by default.

C#

```
HubConnection connection= new HubConnectionBuilder()
    .WithUrl(new Uri("http://127.0.0.1:5000/chathub"))
    .WithAutomaticReconnect()
    .Build();
```

Without any parameters, `WithAutomaticReconnect()` configures the client to wait 0, 2, 10, and 30 seconds respectively before trying each reconnect attempt, stopping after four failed attempts.

Before starting any reconnect attempts, the `HubConnection` will transition to the `HubConnectionState.Reconnecting` state and fire the `Reconnecting` event. This provides an opportunity to warn users that the connection has been lost and to disable UI elements. Non-interactive apps can start queuing or dropping messages.

C#

```
connection.Reconnecting += error =>
{
    Debug.Assert(connection.State == HubConnectionState.Reconnecting);

    // Notify users the connection was lost and the client is reconnecting.
    // Start queuing or dropping messages.

    return Task.CompletedTask;
};
```

If the client successfully reconnects within its first four attempts, the `HubConnection` will transition back to the `Connected` state and fire the `Reconnected` event. This provides an

opportunity to inform users the connection has been reestablished and dequeue any queued messages.

Since the connection looks entirely new to the server, a new `ConnectionId` will be provided to the `Reconnected` event handlers.

> ⚠ **Warning**
>
> The `Reconnected` event handler's `connectionId` parameter will be null if the `HubConnection` was configured to skip negotiation.

C#

```
connection.Reconnected += connectionId =>
{
    Debug.Assert(connection.State == HubConnectionState.Connected);

    // Notify users the connection was reestablished.
    // Start dequeuing messages queued while reconnecting if any.

    return Task.CompletedTask;
};
```

`WithAutomaticReconnect()` won't configure the `HubConnection` to retry initial start failures, so start failures need to be handled manually:

C#

```
public static async Task<bool> ConnectWithRetryAsync(HubConnection
connection, CancellationToken token)
{
    // Keep trying to until we can start or the token is canceled.
    while (true)
    {
        try
        {
            await connection.StartAsync(token);
            Debug.Assert(connection.State == HubConnectionState.Connected);
            return true;
        }
        catch when (token.IsCancellationRequested)
        {
            return false;
        }
        catch
        {
            // Failed to connect, trying again in 5000 ms.
            Debug.Assert(connection.State ==
```

```
HubConnectionState.Disconnected);
            await Task.Delay(5000);
        }
    }
}
```

If the client doesn't successfully reconnect within its first four attempts, the `HubConnection` will transition to the `Disconnected` state and fire the Closed event. This provides an opportunity to attempt to restart the connection manually or inform users the connection has been permanently lost.

C#

```
connection.Closed += error =>
{
    Debug.Assert(connection.State == HubConnectionState.Disconnected);

    // Notify users the connection has been closed or manually try to
restart the connection.

    return Task.CompletedTask;
};
```

In order to configure a custom number of reconnect attempts before disconnecting or change the reconnect timing, `WithAutomaticReconnect` accepts an array of numbers representing the delay in milliseconds to wait before starting each reconnect attempt.

C#

```
HubConnection connection= new HubConnectionBuilder()
    .WithUrl(new Uri("http://127.0.0.1:5000/chathub"))
    .WithAutomaticReconnect(new[] { TimeSpan.Zero, TimeSpan.Zero,
TimeSpan.FromSeconds(10) })
    .Build();

    // .WithAutomaticReconnect(new[] { TimeSpan.Zero,
TimeSpan.FromSeconds(2), TimeSpan.FromSeconds(10), TimeSpan.FromSeconds(30)
}) yields the default behavior.
```

The preceding example configures the `HubConnection` to start attempting reconnects immediately after the connection is lost. This is also true for the default configuration.

If the first reconnect attempt fails, the second reconnect attempt will also start immediately instead of waiting 2 seconds like it would in the default configuration.

If the second reconnect attempt fails, the third reconnect attempt will start in 10 seconds which is again like the default configuration.

The custom behavior then diverges again from the default behavior by stopping after the third reconnect attempt failure. In the default configuration there would be one more reconnect attempt in another 30 seconds.

If you want even more control over the timing and number of automatic reconnect attempts, `WithAutomaticReconnect` accepts an object implementing the `IRetryPolicy` interface, which has a single method named `NextRetryDelay`.

`NextRetryDelay` takes a single argument with the type `RetryContext`. The `RetryContext` has three properties: `PreviousRetryCount`, `ElapsedTime` and `RetryReason`, which are a `long`, a `TimeSpan` and an `Exception` respectively. Before the first reconnect attempt, both `PreviousRetryCount` and `ElapsedTime` will be zero, and the `RetryReason` will be the Exception that caused the connection to be lost. After each failed retry attempt, `PreviousRetryCount` will be incremented by one, `ElapsedTime` will be updated to reflect the amount of time spent reconnecting so far, and the `RetryReason` will be the Exception that caused the last reconnect attempt to fail.

`NextRetryDelay` must return either a TimeSpan representing the time to wait before the next reconnect attempt or `null` if the `HubConnection` should stop reconnecting.

C#

```
public class RandomRetryPolicy : IRetryPolicy
{
    private readonly Random _random = new Random();

    public TimeSpan? NextRetryDelay(RetryContext retryContext)
    {
        // If we've been reconnecting for less than 60 seconds so far,
        // wait between 0 and 10 seconds before the next reconnect attempt.
        if (retryContext.ElapsedTime < TimeSpan.FromSeconds(60))
        {
            return TimeSpan.FromSeconds(_random.NextDouble() * 10);
        }
        else
        {
            // If we've been reconnecting for more than 60 seconds so far,
stop reconnecting.
            return null;
        }
    }
}
```

C#

```
HubConnection connection = new HubConnectionBuilder()
    .WithUrl(new Uri("http://127.0.0.1:5000/chathub"))
```

```
    .WithAutomaticReconnect(new RandomRetryPolicy())
    .Build();
```

Alternatively, you can write code that will reconnect your client manually as demonstrated in Manually reconnect.

Manually reconnect

Use the Closed event to respond to a lost connection. For example, you might want to automate reconnection.

The `Closed` event requires a delegate that returns a `Task`, which allows async code to run without using `async void`. To satisfy the delegate signature in a `Closed` event handler that runs synchronously, return `Task.CompletedTask`:

C#

```
connection.Closed += (error) => {
    // Do your close logic.
    return Task.CompletedTask;
};
```

The main reason for the async support is so you can restart the connection. Starting a connection is an async action.

In a `Closed` handler that restarts the connection, consider waiting for some random delay to prevent overloading the server, as shown in the following example:

C#

```
connection.Closed += async (error) =>
{
    await Task.Delay(new Random().Next(0,5) * 1000);
    await connection.StartAsync();
};
```

Call hub methods from client

`InvokeAsync` calls methods on the hub. Pass the hub method name and any arguments defined in the hub method to `InvokeAsync`. SignalR is asynchronous, so use `async` and `await` when making the calls.

C#

```
await connection.InvokeAsync("SendMessage",
    userTextBox.Text, messageTextBox.Text);
```

The `InvokeAsync` method returns a `Task` which completes when the server method returns. The return value, if any, is provided as the result of the `Task`. Any exceptions thrown by the method on the server produce a faulted `Task`. Use `await` syntax to wait for the server method to complete and `try...catch` syntax to handle errors.

The `SendAsync` method returns a `Task` which completes when the message has been sent to the server. No return value is provided since this `Task` doesn't wait until the server method completes. Any exceptions thrown on the client while sending the message produce a faulted `Task`. Use `await` and `try...catch` syntax to handle send errors.

> **Note**
>
> Calling hub methods from a client is only supported when using the Azure SignalR Service in *Default* mode. For more information, see Frequently Asked Questions (azure-signalr GitHub repository).

Call client methods from hub

Define methods the hub calls using `connection.On` after building, but before starting the connection.

C#

```
connection.On<string, string>("ReceiveMessage", (user, message) =>
{
    this.Dispatcher.Invoke(() =>
    {
        var newMessage = $"{user}: {message}";
        messagesList.Items.Add(newMessage);
    });
});
```

The preceding code in `connection.On` runs when server-side code calls it using the `SendAsync` method.

C#

```
public async Task SendMessage(string user, string message)
{
    await Clients.All.SendAsync("ReceiveMessage", user,message);
}
```

> **Note**
>
> While the hub side of the connection supports strongly-typed messaging, the client must register using the generic method HubConnection.On with the method name. For an example, see Host ASP.NET Core SignalR in background services.

Error handling and logging

Handle errors with a try-catch statement. Inspect the `Exception` object to determine the proper action to take after an error occurs.

C#

```
try
{
    await connection.InvokeAsync("SendMessage",
        userTextBox.Text, messageTextBox.Text);
}
catch (Exception ex)
{
    messagesList.Items.Add(ex.Message);
}
```

Additional resources

- Hubs
- JavaScript client
- Publish to Azure
- Azure SignalR Service serverless documentation

Microsoft.AspNetCore.SignalR.Client Namespace

Reference

Contains types that are used for communicating with a SignalR server.

Classes

Expand table

HubConnection	A connection used to invoke hub methods on a SignalR Server.
HubConnectionBuilder	A builder for configuring HubConnection instances.
HubConnectionBuilderExtensions	Extension methods for IHubConnectionBuilder.
HubConnectionBuilderHttpExtensions	Extension methods for IHubConnectionBuilder.
HubConnectionExtensions	Extension methods for HubConnectionExtensions.
HubConnectionOptions	Configures options for the HubConnection.
RetryContext	The context passed to NextRetryDelay(RetryContext) to help the policy determine how long to wait before the next retry and whether there should be another retry at all.

Interfaces

Expand table

IHubConnectionBuilder	A builder abstraction for configuring HubConnection instances.
IRetryPolicy	An abstraction that controls when the client attempts to reconnect and how many times it does so.

Enums

Expand table

HubConnectionState	Describes the current state of the HubConnection to the server.

Remarks

For more information about the SignalR client, see ASP.NET Core SignalR .NET Client.

Feedback

Was this page helpful? Yes No

ASP.NET Core SignalR Java client

Article • 06/18/2024

By Mikael Mengistu

The Java client enables connecting to an ASP.NET Core SignalR server from Java code, including Android apps. Like the JavaScript client and the .NET client, the Java client enables you to receive and send messages to a hub in real time. The Java client is available in ASP.NET Core 2.2 and later.

The sample Java console app referenced in this article uses the SignalR Java client.

View or download sample code (how to download)

Install the SignalR Java client package

The *signalr-7.0.0* JAR file allows clients to connect to SignalR hubs. To find the latest JAR file version number, see the Maven search results.

If using Gradle, add the following line to the `dependencies` section of your *build.gradle* file:

Gradle

```
implementation 'com.microsoft.signalr:signalr:7.0.0'
```

If using Maven, add the following lines inside the `<dependencies>` element of your `pom.xml` file:

XML

```
<dependency>
    <groupId>com.microsoft.signalr</groupId>
    <artifactId>signalr</artifactId>
    <version>1.0.0</version>
</dependency>
```

Connect to a hub

To establish a `HubConnection`, the `HubConnectionBuilder` should be used. The hub URL and log level can be configured while building a connection. Configure any required

options by calling any of the `HubConnectionBuilder` methods before `build`. Start the connection with `start`.

```
Java

HubConnection hubConnection = HubConnectionBuilder.create(input)
        .build();
```

Call hub methods from client

A call to `send` invokes a hub method. Pass the hub method name and any arguments defined in the hub method to `send`.

```
Java

hubConnection.send("Send", input);
```

> **Note**
>
> Calling hub methods from a client is only supported when using the Azure SignalR Service in *Default* mode. For more information, see Frequently Asked Questions (azure-signalr GitHub repository).

Call client methods from hub

Use `hubConnection.on` to define methods on the client that the hub can call. Define the methods after building but before starting the connection.

```
Java

hubConnection.on("Send", (message) -> {
    System.out.println("New Message: " + message);
}, String.class);
```

Add logging

The SignalR Java client uses the SLF4J library for logging. It's a high-level logging API that allows users of the library to choose their own specific logging implementation by

bringing in a specific logging dependency. The following code snippet shows how to use `java.util.logging` with the SignalR Java client.

Gradle

```
implementation 'org.slf4j:slf4j-jdk14:1.7.25'
```

If you don't configure logging in your dependencies, SLF4J loads a default no-operation logger with the following warning message:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
```

This can safely be ignored.

Android development notes

With regards to Android SDK compatibility for the SignalR client features, consider the following items when specifying your target Android SDK version:

- The SignalR Java Client will run on Android API Level 16 and later.
- Connecting through the Azure SignalR Service will require Android API Level 20 and later because the Azure SignalR Service requires TLS 1.2 and doesn't support SHA-1-based cipher suites. Android added support for SHA-256 (and above) cipher suites in API Level 20.

Configure bearer token authentication

In the SignalR Java client, you can configure a bearer token to use for authentication by providing an "access token factory" to the HttpHubConnectionBuilder. Use withAccessTokenFactory to provide an RxJava Single<String>. With a call to Single.defer, you can write logic to produce access tokens for your client.

Java

```
HubConnection hubConnection = HubConnectionBuilder.create("YOUR HUB URL
HERE")
    .withAccessTokenProvider(Single.defer(() -> {
        // Your logic here.
```

```
        return Single.just("An Access Token");
    })).build();
```

Passing Class information in Java

When calling the `on`, `invoke`, or `stream` methods of `HubConnection` in the Java client, users should pass a `Type` object rather than a `Class<?>` object to describe any generic `Object` passed to the method. A `Type` can be acquired using the provided `TypeReference` class. For example, using a custom generic class named `Foo<T>`, the following code gets the `Type`:

Java

```
Type fooType = new TypeReference<Foo<String>>() { }).getType();
```

For non-generics, such as primitives or other non-parameterized types like `String`, you can simply use the built-in `.class`.

When calling one of these methods with one or more object types, use the generics syntax when invoking the method. For example, when registering an `on` handler for a method named `func`, which takes as arguments a String and a `Foo<String>` object, use the following code to set an action to print the arguments:

Java

```
hubConnection.<String, Foo<String>>on("func", (param1, param2) ->{
    System.out.println(param1);
    System.out.println(param2);
}, String.class, fooType);
```

This convention is necessary because we can not retrieve complete information about complex types with the `Object.getClass` method due to type erasure in Java. For example, calling `getClass` on an `ArrayList<String>` would not return `Class<ArrayList<String>>`, but rather `Class<ArrayList>`, which does not give the deserializer enough information to correctly deserialize an incoming message. The same is true for custom objects.

Known limitations

- Transport fallback and the Server Sent Events transport aren't supported.

Additional resources

- Java API reference
- Use hubs in ASP.NET Core SignalR
- ASP.NET Core SignalR JavaScript client
- Publish an ASP.NET Core SignalR app to Azure App Service
- Azure SignalR Service serverless documentation

com.microsoft.signalr

Reference

Package: com.microsoft.signalr

Maven Artifact: com.microsoft.signalr:signalr:5.0.10

This package contains the classes for SignalR Java client.

Classes

Expand table

CancelInvocationMessage	
CloseMessage	
CompletionMessage	
HttpHubConnectionBuilder	A builder for configuring HubConnection instances.
HubConnection	A connection used to invoke hub methods on a SignalR Server.
HubConnectionBuilder	A builder for configuring HubConnection instances.
HubException	An exception thrown when the server fails to invoke a Hub method.
HubMessage	A base class for hub messages.
InvocationBindingFailureMessage	
InvocationMessage	
PingMessage	
StreamBindingFailureMessage	
StreamInvocationMessage	
StreamItem	
Subscription	Represents the registration of a handler for a client method.
TypeReference<T>	A utility for getting a Java Type from a literal generic Class.
UserAgentHelper	

Interfaces

Expand table

Action	A callback that takes no parameters.
Action1<T1>	A callback that takes one parameter.
Action2<T1,T2>	A callback that takes two parameters.
Action3<T1,T2,T3>	A callback that takes three parameters.
Action4<T1,T2,T3,T4>	A callback that takes four parameters.
Action5<T1,T2,T3,T4,T5>	A callback that takes five parameters.
Action6<T1,T2,T3,T4,T5,T6>	A callback that takes six parameters.
Action7<T1,T2,T3,T4,T5,T6,T7>	A callback that takes seven parameters.
Action8<T1,T2,T3,T4,T5,T6,T7,T8>	A callback that takes eight parameters.
HubProtocol	A protocol abstraction for communicating with SignalR hubs.
InvocationBinder	An abstraction for passing around information about method signatures.
OnClosedCallback	A callback to create and register on a HubConnections On Closed method.

Enums

Expand table

HubConnectionState	Indicates the state of the HubConnection.
HubMessageType	
TransportEnum	Used to specify the transport the client will use.

Feedback

Was this page helpful? Yes No

ASP.NET Core SignalR JavaScript client

Article • 06/18/2024

By Rachel Appel

The ASP.NET Core SignalR JavaScript client library enables developers to call server-side SignalR hub code.

Install the SignalR client package

The SignalR JavaScript client library is delivered as an npm package. The following sections outline different ways to install the client library.

Install with npm

Visual Studio

Run the following commands from **Package Manager Console**:

Bash

```
npm init -y
npm install @microsoft/signalr
```

npm installs the package contents in the *node_modules\@microsoft\signalr\dist\browser* folder. Create the *wwwroot/lib/signalr* folder. Copy the `signalr.js` file to the *wwwroot/lib/signalr* folder.

Reference the SignalR JavaScript client in the `<script>` element. For example:

HTML

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

Use a Content Delivery Network (CDN)

To use the client library without the npm prerequisite, reference a CDN-hosted copy of the client library. For example:

HTML

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-
signalr/6.0.1/signalr.js"></script>
```

The client library is available on the following CDNs:

- cdnjs
- jsDelivr
- unpkg

Install with LibMan

LibMan can be used to install specific client library files from the CDN-hosted client library. For example, only add the minified JavaScript file to the project. For details on that approach, see Add the SignalR client library.

Connect to a hub

The following code creates and starts a connection. The hub's name is case insensitive:

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();

async function start() {
    try {
        await connection.start();
        console.log("SignalR Connected.");
    } catch (err) {
        console.log(err);
        setTimeout(start, 5000);
    }
};

connection.onclose(async () => {
    await start();
});

// Start the connection.
start();
```

Cross-origin connections (CORS)

Typically, browsers load connections from the same domain as the requested page. However, there are occasions when a connection to another domain is required.

When making cross domain requests, the client code ***must*** use an absolute URL instead of a relative URL. For cross domain requests, change `.withUrl("/chathub")` to `.withUrl("https://{App domain name}/chathub")`.

To prevent a malicious site from reading sensitive data from another site, cross-origin connections are disabled by default. To allow a cross-origin request, enable CORS:

C#

```
using SignalRChat.Hubs;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddSignalR();

builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(
        builder =>
        {
            builder.WithOrigins("https://example.com")
                .AllowAnyHeader()
                .WithMethods("GET", "POST")
                .AllowCredentials();
        });
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

app.UseRouting();

app.UseAuthorization();

// UseCors must be called before MapHub.
app.UseCors();

app.MapRazorPages();
app.MapHub<ChatHub>("/chatHub");
```

```
app.Run();
```

UseCors must be called before calling MapHub.

Call hub methods from the client

JavaScript clients call public methods on hubs via the invoke method of the HubConnection. The `invoke` method accepts:

- The name of the hub method.
- Any arguments defined in the hub method.

In the following highlighted code, the method name on the hub is `SendMessage`. The second and third arguments passed to `invoke` map to the hub method's `user` and `message` arguments:

JavaScript

```
try {
    await connection.invoke("SendMessage", user, message);
} catch (err) {
    console.error(err);
}
```

Calling hub methods from a client is only supported when using the ***Azure SignalR Service in Default*** mode. For more information, see Frequently Asked Questions (azure-signalr GitHub repository).

The `invoke` method returns a JavaScript Promise. The `Promise` is resolved with the return value (if any) when the method on the server returns. If the method on the server throws an error, the `Promise` is rejected with the error message. Use `async` and `await` or the `Promise`'s `then` and `catch` methods to handle these cases.

JavaScript clients can also call public methods on hubs via the send method of the `HubConnection`. Unlike the `invoke` method, the `send` method doesn't wait for a response from the server. The `send` method returns a JavaScript `Promise`. The `Promise` is resolved when the message has been sent to the server. If there is an error sending the message, the `Promise` is rejected with the error message. Use `async` and `await` or the `Promise`'s `then` and `catch` methods to handle these cases.

Using `send` ***doesn't*** wait until the server has received the message. Consequently, it's not possible to return data or errors from the server.

Call client methods from the hub

To receive messages from the hub, define a method using the on method of the `HubConnection`.

- The name of the JavaScript client method.
- Arguments the hub passes to the method.

In the following example, the method name is `ReceiveMessage`. The argument names are `user` and `message`:

JavaScript

```
connection.on("ReceiveMessage", (user, message) => {
    const li = document.createElement("li");
    li.textContent = `${user}: ${message}`;
    document.getElementById("messageList").appendChild(li);
});
```

The preceding code in `connection.on` runs when server-side code calls it using the SendAsync method:

C#

```
using Microsoft.AspNetCore.SignalR;
namespace SignalRChat.Hubs;

public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
    }
}
```

SignalR determines which client method to call by matching the method name and arguments defined in `SendAsync` and `connection.on`.

A **best practice** is to call the start method on the `HubConnection` after `on`. Doing so ensures the handlers are registered before any messages are received.

Error handling and logging

Use `console.error` to output errors to the browser's console when the client can't connect or send a message:

JavaScript

```
try {
    await connection.invoke("SendMessage", user, message);
} catch (err) {
    console.error(err);
}
```

Set up client-side log tracing by passing a logger and type of event to log when the connection is made. Messages are logged with the specified log level and higher. Available log levels are as follows:

- `signalR.LogLevel.Error`: Error messages. Logs `Error` messages only.
- `signalR.LogLevel.Warning`: Warning messages about potential errors. Logs `Warning`, and `Error` messages.
- `signalR.LogLevel.Information`: Status messages without errors. Logs `Information`, `Warning`, and `Error` messages.
- `signalR.LogLevel.Trace`: Trace messages. Logs everything, including data transported between hub and client.

Use the configureLogging method on HubConnectionBuilder to configure the log level. Messages are logged to the browser console:

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

Reconnect clients

Automatically reconnect

The JavaScript client for SignalR can be configured to automatically reconnect using the WithAutomaticReconnect method on HubConnectionBuilder. It won't automatically reconnect by default.

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
```

```
    .withAutomaticReconnect()
    .build();
```

Without any parameters, WithAutomaticReconnect configures the client to wait 0, 2, 10, and 30 seconds respectively before trying each reconnect attempt. After four failed attempts, it stops trying to reconnect.

Before starting any reconnect attempts, the `HubConnection`:

- Transitions to the HubConnectionState.Reconnecting state and fires its `onreconnecting` callbacks.
- Doesn't transition to the `Disconnected` state and trigger its `onclose` callbacks like a `HubConnection` without automatic reconnect configured.

The reconnect approach provides an opportunity to:

- Warn users that the connection has been lost.
- Disable UI elements.

JavaScript

```
connection.onreconnecting(error => {
    console.assert(connection.state ===
signalR.HubConnectionState.Reconnecting);

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

    const li = document.createElement("li");
    li.textContent = `Connection lost due to error "${error}".
Reconnecting.`;
    document.getElementById("messageList").appendChild(li);
});
```

If the client successfully reconnects within its first four attempts, the `HubConnection` transitions back to the `Connected` state and fire its `onreconnected` callbacks. This provides an opportunity to inform users the connection has been reestablished.

Since the connection looks entirely new to the server, a new `connectionId` is provided to the `onreconnected` callback.

The `onreconnected` callback's `connectionId` parameter is ***undefined*** if the `HubConnection` is configured to skip negotiation.

JavaScript

```
connection.onreconnected(connectionId => {
    console.assert(connection.state ===
signalR.HubConnectionState.Connected);

    document.getElementById("messageInput").disabled = false;

    const li = document.createElement("li");
    li.textContent = `Connection reestablished. Connected with connectionId
"${connectionId}".`;
    document.getElementById("messageList").appendChild(li);
});
```

`withAutomaticReconnect` won't configure the `HubConnection` to retry initial start failures, so start failures need to be handled manually:

JavaScript

```
async function start() {
    try {
        await connection.start();
        console.assert(connection.state ===
signalR.HubConnectionState.Connected);
        console.log("SignalR Connected.");
    } catch (err) {
        console.assert(connection.state ===
signalR.HubConnectionState.Disconnected);
        console.log(err);
        setTimeout(() => start(), 5000);
    }
};
```

If the client doesn't successfully reconnect within its first four attempts, the `HubConnection` transitions to the `Disconnected` state and fires its onclose callbacks. This provides an opportunity to inform users:

- The connection has been permanently lost.
- Try refreshing the page:

JavaScript

```
connection.onclose(error => {
    console.assert(connection.state ===
signalR.HubConnectionState.Disconnected);

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

    const li = document.createElement("li");
    li.textContent = `Connection closed due to error "${error}". Try
refreshing this page to restart the connection.`;
```

```
    document.getElementById("messageList").appendChild(li);
});
```

In order to configure a custom number of reconnect attempts before disconnecting or change the reconnect timing, `withAutomaticReconnect` accepts an array of numbers representing the delay in milliseconds to wait before starting each reconnect attempt.

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect([0, 0, 10000])
    .build();

    // .withAutomaticReconnect([0, 2000, 10000, 30000]) yields the default
behavior
```

The preceding example configures the `HubConnection` to start attempting reconnects immediately after the connection is lost. The default configuration also waits zero seconds to attempt reconnecting.

If the first reconnect attempt fails, the second reconnect attempt also starts immediately instead of waiting 2 seconds using the default configuration.

If the second reconnect attempt fails, the third reconnect attempt start in 10 seconds which is the same as the default configuration.

The configured reconnection timing differs from the default behavior by stopping after the third reconnect attempt failure instead of trying one more reconnect attempt in another 30 seconds.

For more control over the timing and number of automatic reconnect attempts, `withAutomaticReconnect` accepts an object implementing the `IRetryPolicy` interface, which has a single method named `nextRetryDelayInMilliseconds`.

`nextRetryDelayInMilliseconds` takes a single argument with the type `RetryContext`. The `RetryContext` has three properties: `previousRetryCount`, `elapsedMilliseconds` and `retryReason` which are a `number`, a `number` and an `Error` respectively. Before the first reconnect attempt, both `previousRetryCount` and `elapsedMilliseconds` will be zero, and the `retryReason` will be the Error that caused the connection to be lost. After each failed retry attempt, `previousRetryCount` will be incremented by one, `elapsedMilliseconds` will be updated to reflect the amount of time spent reconnecting so far in milliseconds, and the `retryReason` will be the Error that caused the last reconnect attempt to fail.

`nextRetryDelayInMilliseconds` must return either a number representing the number of milliseconds to wait before the next reconnect attempt or `null` if the `HubConnection` should stop reconnecting.

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect({
        nextRetryDelayInMilliseconds: retryContext => {
            if (retryContext.elapsedMilliseconds < 60000) {
                // If we've been reconnecting for less than 60 seconds so
far,
                // wait between 0 and 10 seconds before the next reconnect
attempt.
                return Math.random() * 10000;
            } else {
                // If we've been reconnecting for more than 60 seconds so
far, stop reconnecting.
                return null;
            }
        }
    })
    .build();
```

Alternatively, code can be written that reconnects the client manually as demonstrated in the following section.

Manually reconnect

The following code demonstrates a typical manual reconnection approach:

1. A function (in this case, the `start` function) is created to start the connection.
2. Call the `start` function in the connection's `onclose` event handler.

JavaScript

```
async function start() {
    try {
        await connection.start();
        console.log("SignalR Connected.");
    } catch (err) {
        console.log(err);
        setTimeout(start, 5000);
    }
};

connection.onclose(async () => {
```

```
    await start();
});
```

Production implementations typically use an exponential back-off or retry a specified number of times.

Browser sleeping tab

Some browsers have a tab freezing or sleeping feature to reduce computer resource usage for inactive tabs. This can cause SignalR connections to close and may result in an unwanted user experience. Browsers use heuristics to figure out if a tab should be put to sleep, such as:

- Playing audio
- Holding a web lock
- Holding an `IndexedDB` lock
- Being connected to a USB device
- Capturing video or audio
- Being mirrored
- Capturing a window or display

Browser heuristics may change over time and can differ between browsers. Check the support matrix and figure out what method works best for your scenarios.

To avoid putting an app to sleep, the app should trigger one of the heuristics that the browser uses.

The following code example shows how to use a Web Lock to keep a tab awake and avoid an unexpected connection closure.

JavaScript

```
var lockResolver;
if (navigator && navigator.locks && navigator.locks.request) {
    const promise = new Promise((res) => {
        lockResolver = res;
    });

    navigator.locks.request('unique_lock_name', { mode: "shared" }, () => {
        return promise;
    });
}
```

For the preceding code example:

- Web Locks are experimental. The conditional check confirms that the browser supports Web Locks.
- The promise resolver, `lockResolver`, is stored so that the lock can be released when it's acceptable for the tab to sleep.
- When closing the connection, the lock is released by calling `lockResolver()`. When the lock is released, the tab is allowed to sleep.

Additional resources

- View or download sample code (how to download)
- JavaScript API reference
- JavaScript tutorial
- WebPack and TypeScript tutorial
- Hubs
- .NET client
- Publish to Azure
- Cross-Origin Requests (CORS)
- Azure SignalR Service serverless documentation
- Troubleshoot connection errors

@microsoft/signalr package

Reference

Classes

Expand table

DefaultHttpClient	Default implementation of HttpClient.
AbortError	Error thrown when an action is aborted.
HttpError	Error thrown when an HTTP request fails.
TimeoutError	Error thrown when a timeout elapses.
FetchHttpClient	
HeaderNames	
HttpClient	Abstraction over an HTTP client. This class provides an abstraction over an HTTP client so that a different implementation can be provided on different platforms.
HttpResponse	Represents an HTTP response.
HubConnection	Represents a connection to a SignalR Hub.
HubConnectionBuilder	A builder for configuring HubConnection instances.
JsonHubProtocol	Implements the JSON Hub Protocol.
NullLogger	A logger that does nothing when log messages are sent to it.
Subject	Stream implementation to stream items to the server.
XhrHttpClient	

Interfaces

Expand table

AbortSignal	Represents a signal that can be monitored to determine if a request has been aborted.

HttpRequest	Represents an HTTP request.
IHttpConnection Options	Options provided to the 'withUrl' method on HubConnectionBuilder to configure options for the HTTP-based transports.
CancelInvocation Message	A hub message sent to request that a streaming invocation be canceled.
CloseMessage	A hub message indicating that the sender is closing the connection. If error is defined, the sender is closing the connection due to an error.
Completion Message	A hub message representing the result of an invocation.
HubInvocation Message	Defines properties common to all Hub messages relating to a specific invocation.
HubMessageBase	Defines properties common to all Hub messages.
IHubProtocol	A protocol abstraction for communicating with SignalR Hubs.
Invocation Message	A hub message representing a non-streaming invocation.
MessageHeaders	Defines a dictionary of string keys and string values representing headers attached to a Hub message.
PingMessage	A hub message indicating that the sender is still active.
StreamInvocation Message	A hub message representing a streaming invocation.
StreamItem Message	A hub message representing a single item produced as part of a result stream.
ILogger	An abstraction that provides a sink for diagnostic messages.
IRetryPolicy	An abstraction that controls when the client attempts to reconnect and how many times it does so.
RetryContext	
ITransport	An abstraction over the behavior of transports. This is designed to support the framework and not intended for use by applications.
IStreamResult	Defines the result of a streaming hub method.
IStreamSubscriber	Defines the expected type for a receiver of results streamed by the server.
ISubscription	An interface that allows an IStreamSubscriber to be disconnected from a stream.

Type Aliases

Expand table

HubMessage	Union type of all known Hub messages.

Enums

Expand table

HubConnectionState	Describes the current state of the HubConnection to the server.
MessageType	Defines the type of a Hub Message.
LogLevel	Indicates the severity of a log message. Log Levels are ordered in increasing severity. So `Debug` is more severe than `Trace`, etc.
HttpTransportType	Specifies a specific HTTP transport type.
TransferFormat	Specifies the transfer format for a connection.

ASP.NET Core SignalR hosting and scaling

Article • 09/17/2024

By Andrew Stanton-Nurse, Brady Gaster, and Tom Dykstra

This article explains hosting and scaling considerations for high-traffic apps that use ASP.NET Core SignalR.

Sticky Sessions

SignalR requires that all HTTP requests for a specific connection be handled by the same server process. When SignalR is running on a server farm (multiple servers), "sticky sessions" must be used. "Sticky sessions" are also called session affinity. Azure App Service uses Application Request Routing (ARR) to route requests. Enabling the "Session affinity" (ARR Affinity) setting in your Azure App Service enables "sticky sessions." The only circumstances in which sticky sessions aren't required for the app are:

1. When hosting on a single server in a single process.
2. When using the Azure SignalR Service (sticky sessions are enabled for the service, not the app).
3. When all clients are configured to **only** use WebSockets, **and** the SkipNegotiation setting is enabled in the client configuration.

In all other circumstances (including when the Redis backplane is used), the server environment must be configured for sticky sessions.

For guidance on configuring Azure App Service for SignalR, see Publish an ASP.NET Core SignalR app to Azure App Service. For guidance on configuring sticky sessions for Blazor apps that use the Azure SignalR Service, see Host and deploy ASP.NET Core server-side Blazor apps.

TCP connection resources

The number of concurrent TCP connections that a web server can support is limited. Standard HTTP clients use *ephemeral* connections. These connections can be closed when the client goes idle and reopened later. On the other hand, a SignalR connection is *persistent*. SignalR connections stay open even when the client goes idle. In a high-traffic app that serves many clients, these persistent connections can cause servers to hit their maximum number of connections.

Persistent connections also consume some additional memory, to track each connection.

The heavy use of connection-related resources by SignalR can affect other web apps that are hosted on the same server. When SignalR opens and holds the last available TCP connections, other web apps on the same server also have no more connections available to them.

If a server runs out of connections, you'll see random socket errors and connection reset errors. For example:

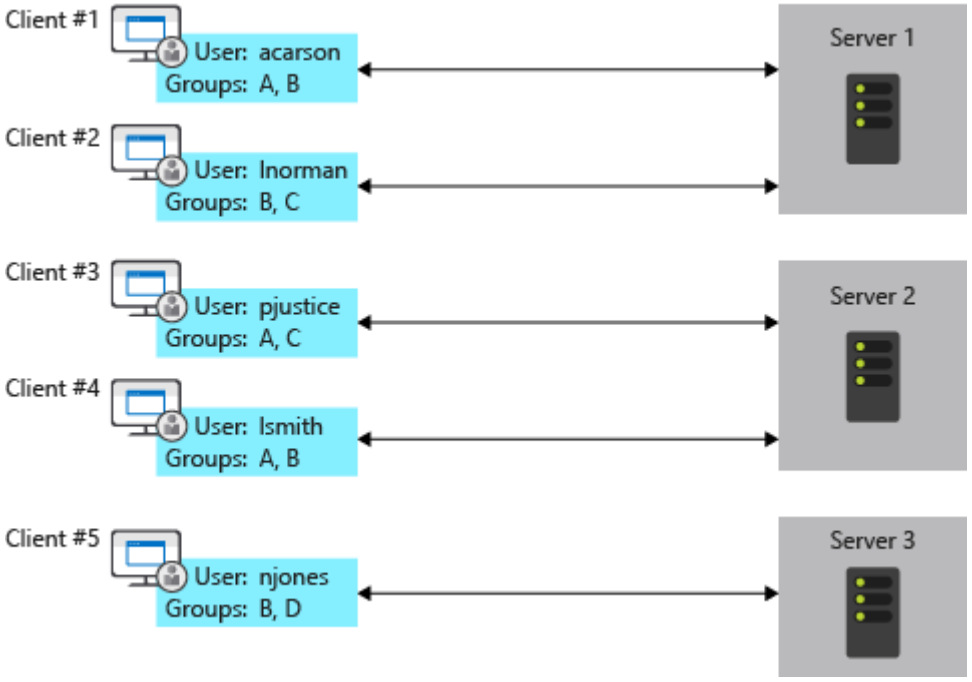
```
An attempt was made to access a socket in a way forbidden by its access permissions...
```

To keep SignalR resource usage from causing errors in other web apps, run SignalR on different servers than your other web apps.

To keep SignalR resource usage from causing errors in a SignalR app, scale out to limit the number of connections a server has to handle.

Scale out

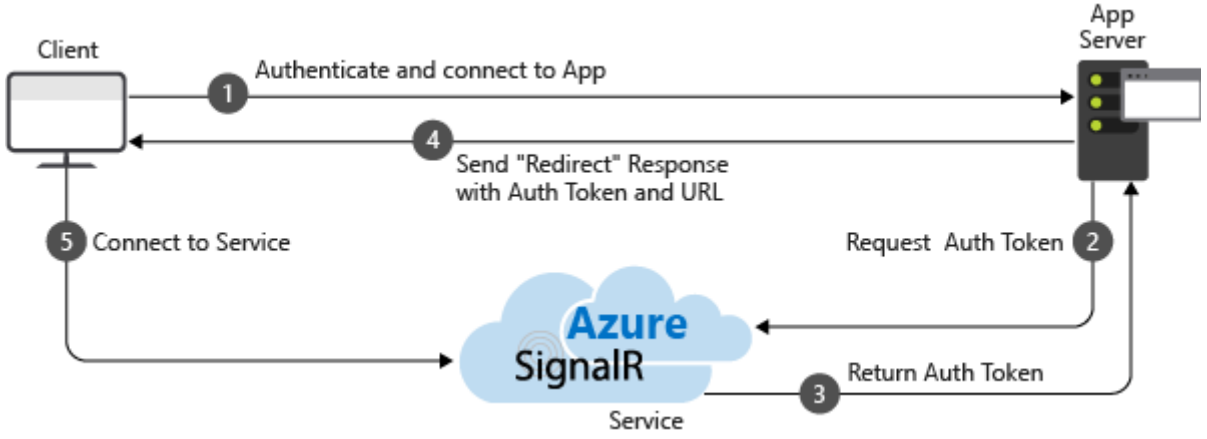
An app that uses SignalR needs to keep track of all its connections, which creates problems for a server farm. Add a server, and it gets new connections that the other servers don't know about. For example, SignalR on each server in the following diagram is unaware of the connections on the other servers. When SignalR on one of the servers wants to send a message to all clients, the message only goes to the clients connected to that server.



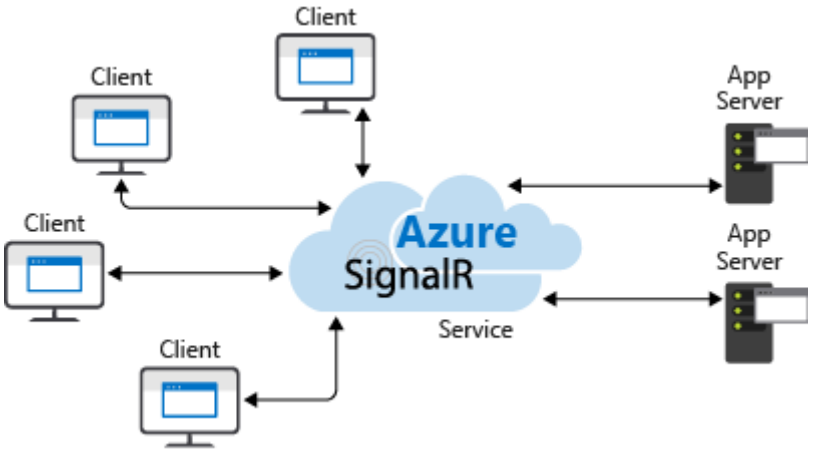
The options for solving this problem are the Azure SignalR Service and Redis backplane.

Azure SignalR Service

The Azure SignalR Service functions as a proxy for real-time traffic and doubles as a backplane when the app is scaled out across multiple servers. Each time a client initiates a connection to the server, the client is redirected to connect to the service. The process is illustrated by the following diagram:



The result is that the service manages all of the client connections, while each server needs only a small constant number of connections to the service, as shown in the following diagram:



This approach to scale-out has several advantages over the Redis backplane alternative:

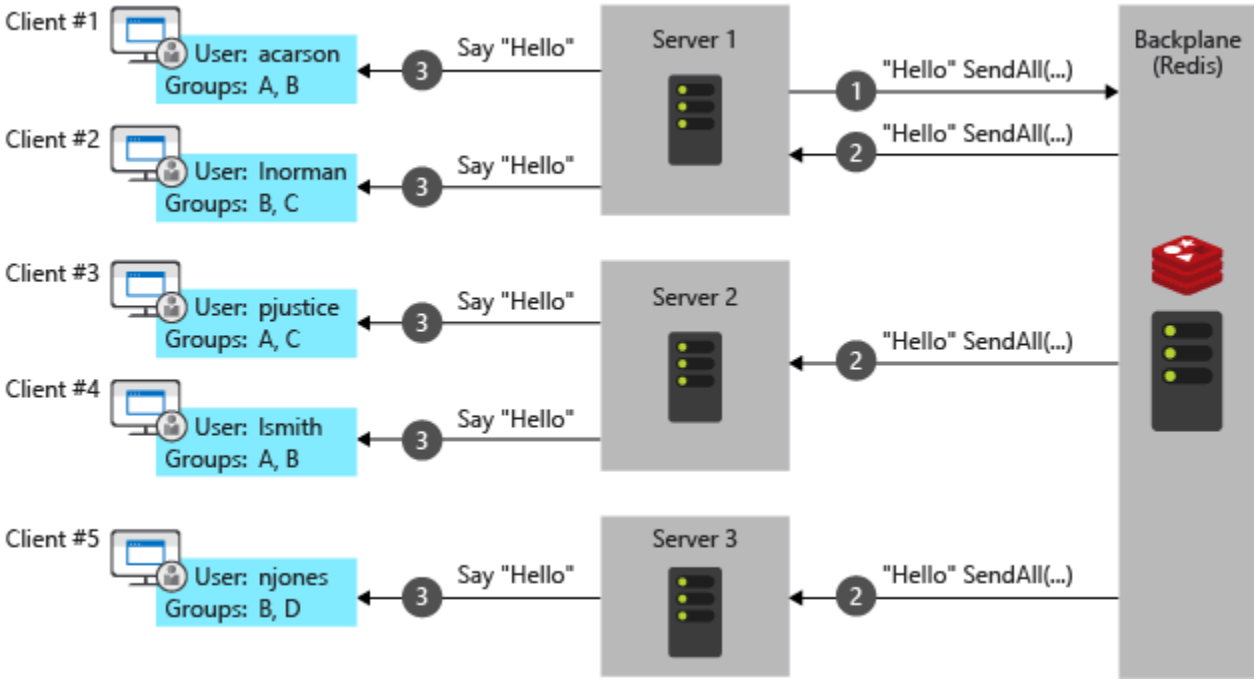
- Sticky sessions, also known as client affinity, is not required, because clients are immediately redirected to the Azure SignalR Service when they connect.
- A SignalR app can scale out based on the number of messages sent, while the Azure SignalR Service scales to handle any number of connections. For example, there could be thousands of clients, but if only a few messages per second are sent, the SignalR app won't need to scale out to multiple servers just to handle the connections themselves.
- A SignalR app won't use significantly more connection resources than a web app without SignalR.

For these reasons, we recommend the Azure SignalR Service for all ASP.NET Core SignalR apps hosted on Azure, including App Service, VMs, and containers.

For more information see the Azure SignalR Service documentation.

Redis backplane

Redis is an in-memory key-value store that supports a messaging system with a publish/subscribe model. The SignalR Redis backplane uses the pub/sub feature to forward messages to other servers. When a client makes a connection, the connection information is passed to the backplane. When a server wants to send a message to all clients, it sends to the backplane. The backplane knows all connected clients and which servers they're on. It sends the message to all clients via their respective servers. This process is illustrated in the following diagram:



The Redis backplane is the recommended scale-out approach for apps hosted on your own infrastructure. If there is significant connection latency between your data center and an Azure data center, Azure SignalR Service may not be a practical option for on-premises apps with low latency or high throughput requirements.

The Azure SignalR Service advantages noted earlier are disadvantages for the Redis backplane:

- Sticky sessions, also known as client affinity, is required, except when **both** of the following are true:
  - All clients are configured to **only** use WebSockets.
  - The SkipNegotiation setting is enabled in the client configuration. Once a connection is initiated on a server, the connection has to stay on that server.
- A SignalR app must scale out based on number of clients even if few messages are being sent.
- A SignalR app uses significantly more connection resources than a web app without SignalR.

IIS limitations on Windows client OS

Windows 10 and Windows 8.x are client operating systems. IIS on client operating systems has a limit of 10 concurrent connections. SignalR's connections are:

- Transient and frequently re-established.
- **Not** disposed immediately when no longer used.

The preceding conditions make it likely to hit the 10 connection limit on a client OS. When a client OS is used for development, we recommend:

- Avoid IIS.
- Use Kestrel or IIS Express as deployment targets.

Linux with Nginx

The following contains the minimum required settings to enable WebSockets, ServerSentEvents, and LongPolling for SignalR:

nginx

```
http {
  map $http_connection $connection_upgrade {
    "~*Upgrade" $http_connection;
    default keep-alive;
  }

  server {
    listen 80;
    server_name example.com *.example.com;

    # Configure the SignalR Endpoint
    location /hubroute {
      # App server url
      proxy_pass http://localhost:5000;

      # Configuration for WebSockets
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection $connection_upgrade;
      proxy_cache off;
      # WebSockets were implemented after http/1.0
      proxy_http_version 1.1;

      # Configuration for ServerSentEvents
      proxy_buffering off;

      # Configuration for LongPolling or if your KeepAliveInterval is longer
than 60 seconds
      proxy_read_timeout 100s;

      proxy_set_header Host $host;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header X-Forwarded-Proto $scheme;
    }
  }
}
```

When multiple backend servers are used, sticky sessions must be added to prevent SignalR connections from switching servers when connecting. There are multiple ways to

add sticky sessions in Nginx. Two approaches are shown below depending on what you have available.

The following is added in addition to the previous configuration. In the following examples, `backend` is the name of the group of servers.

With Nginx Open Source, use `ip_hash` to route connections to a server based on the client's IP address:

nginx

```
http {
  upstream backend {
    # App server 1
    server localhost:5000;
    # App server 2
    server localhost:5002;

    ip_hash;
  }
}
```

With Nginx Plus, use `sticky` to add a cookie to requests and pin the user's requests to a server:

nginx

```
http {
  upstream backend {
    # App server 1
    server localhost:5000;
    # App server 2
    server localhost:5002;

    sticky cookie srv_id expires=max domain=.example.com path=/ httponly;
  }
}
```

Finally, change `proxy_pass http://localhost:5000` in the `server` section to `proxy_pass http://backend`.

For more information on WebSockets over Nginx, see NGINX as a WebSocket Proxy.

For more information on load balancing and sticky sessions, see NGINX load balancing.

For more information about ASP.NET Core with Nginx see the following article:

- Host ASP.NET Core on Linux with Nginx

Third-party SignalR backplane providers

- NCache
- Orleans
- Rebus
- SQL Server

Next steps

For more information, see the following resources:

- Azure SignalR Service documentation
- Set up a Redis backplane

Publish an ASP.NET Core SignalR app to Azure App Service

Article • 09/17/2024

By Brady Gaster

Azure App Service is a Microsoft cloud computing platform service for hosting web apps, including ASP.NET Core.

> **Note**
>
> This article refers to publishing an ASP.NET Core SignalR app from Visual Studio. For more information, see SignalR service for Azure.

Publish the app

This article covers publishing using the tools in Visual Studio. Visual Studio Code users can use Azure CLI commands to publish apps to Azure. For more information, see Publish an ASP.NET Core app to Azure with command line tools.

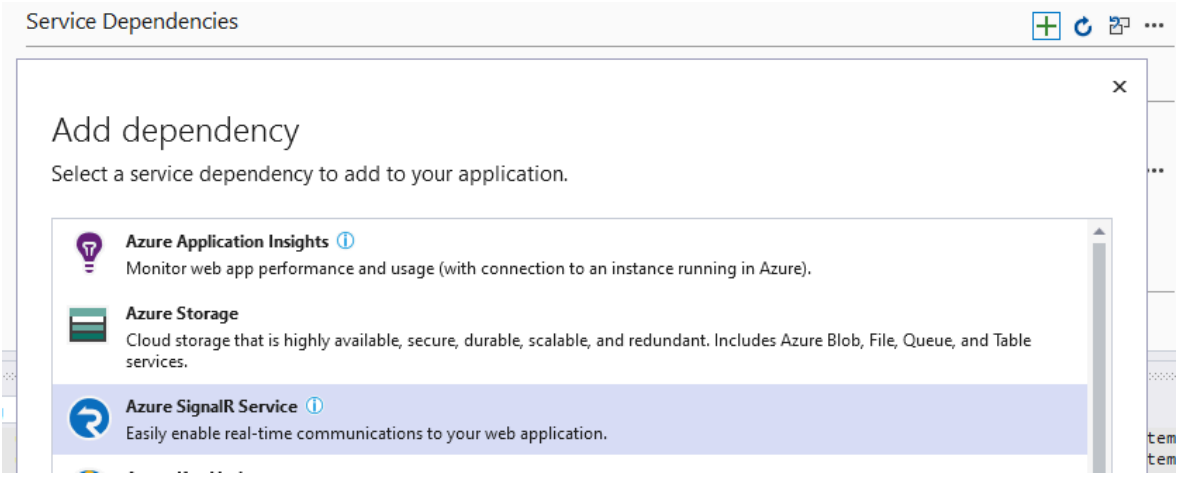
1. Right-click on the project in **Solution Explorer** and select **Publish**.
2. Confirm that **App Service** and **Create new** are selected in the **Pick a publish target** dialog.
3. Select **Create Profile** from the **Publish** button drop down.

   Enter the information described in the following table in the **Create App Service** dialog and select **Create**.

Expand table

Item	Description
Name	Unique name of the app.
Subscription	Azure subscription that the app uses.
Resource Group	Group of related resources to which the app belongs.
Hosting Plan	Pricing plan for the web app.

4. Select **Azure SignalR Service** in the **Service Dependencies** section. Select the + button:



5. In the **Azure SignalR Service** dialog, select **Create a new Azure SignalR Service instance**.

6. Provide a **Name**, **Resource Group**, and **Location**. Return to the **Azure SignalR Service** dialog and select **Add**.

Visual Studio completes the following tasks:

- Creates a Publish Profile containing publish settings.
- Creates an *Azure Web App* with the provided details.
- Publishes the app.
- Launches a browser, which loads the web app.

The format of the app's URL is `{APP SERVICE NAME}.azurewebsites.net`. For example, an app named `SignalRChatApp` has a URL of `https://signalrchatapp.azurewebsites.net`.

If an HTTP *502.2 - Bad Gateway* error occurs when deploying an app that targets a preview .NET Core release, see Deploy ASP.NET Core preview release to Azure App Service to resolve it.

Configure the app in Azure App Service

> **Note**
>
> *This section only applies to apps not using the Azure SignalR Service.*
>
> If the app uses the Azure SignalR Service, the App Service doesn't require the configuration of WebSockets and session affinity, also called Application Request

Routing (ARR) affinity, described in this section. Clients connect their WebSockets to the Azure SignalR Service, not directly to the app.

For apps hosted without the Azure SignalR Service, enable:

- WebSockets to allow the WebSockets transport to function. The default setting is **Off**.
- Session affinity (ARR affinity) to route requests from a user back to the same App Service instance. The default setting is **On**.

1. In the Azure portal, navigate to the web app in **App Services**.
2. Open **Settings** > **Configuration**.
3. Set **Web sockets** to **On**.
4. Verify that **Session affinity** is set to **On**.

App Service Plan limits

WebSockets and other transports are limited based on the App Service Plan selected. For more information, see the *Azure Cloud Services limits* and *App Service limits* sections of the Azure subscription and service limits, quotas, and constraints article.

Additional resources

- What is Azure SignalR Service?
- Overview of ASP.NET Core SignalR
- Host and deploy ASP.NET Core
- Publish an ASP.NET Core app to Azure with Visual Studio
- Publish an ASP.NET Core app to Azure with command line tools
- Host and deploy ASP.NET Core Preview apps on Azure

Set up a Redis backplane for ASP.NET Core SignalR scale-out

Article • 11/04/2024

By Andrew Stanton-Nurse, Brady Gaster, and Tom Dykstra.

This article explains SignalR-specific aspects of setting up a Redis server to use for scaling out an ASP.NET Core SignalR app.

> ⚠ **Warning**
>
> This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

Set up a Redis backplane

- Deploy a Redis server.

  > ⓘ **Important**
  >
  > For production use, a Redis backplane is recommended only when it runs in the same data center as the SignalR app. Otherwise, network latency degrades performance. If your SignalR app is running in the Azure cloud, we recommend Azure SignalR Service instead of a Redis backplane.

  For more information, see the following resources:
  - ASP.NET Core SignalR production hosting and scaling
  - Redis documentation
  - Azure Redis Cache documentation

- In the SignalR app, install the following NuGet package:
  - `Microsoft.AspNetCore.SignalR.StackExchangeRedis`

- Call AddStackExchangeRedis by adding the following line before the line that calls `builder.Build()`) in the `Program.cs` file.

  ```
  C#

  builder.Services.AddSignalR().AddStackExchangeRedis("
  <your_Redis_connection_string>");
  ```

- Configure options as needed:

  Most options can be set in the connection string or in the ConfigurationOptions object. Options specified in `ConfigurationOptions` override the ones set in the connection string.

  The following example shows how to set options in the `ConfigurationOptions` object. This example adds a channel prefix so that multiple apps can share the same Redis instance, as explained in the following step.

  ```
  C#

  builder.Services.AddSignalR()
    .AddStackExchangeRedis(connectionString, options => {
        options.Configuration.ChannelPrefix =
  RedisChannel.Literal("MyApp");
    });
  ```

  In the preceding code, `options.Configuration` is initialized with whatever was specified in the connection string.

  For information about Redis options, see the StackExchange Redis documentation.

- If you're using one Redis server for multiple SignalR apps, use a different channel prefix for each SignalR app.

  Setting a channel prefix isolates one SignalR app from others that use different channel prefixes. If you don't assign different prefixes, a message sent from one app to all of its own clients will go to all clients of all apps that use the Redis server as a backplane.

- Configure your server farm load balancing software for sticky sessions. Here are some examples of documentation on how to do that:
  - IIS
  - HAProxy
  - Nginx

Redis server errors

When a Redis server goes down, SignalR throws exceptions that indicate messages won't be delivered. Some typical exception messages:

- *Failed writing message*
- *Failed to invoke hub method 'MethodName'*
- *Connection to Redis failed*

SignalR doesn't buffer messages to send them when the server comes back up. Any messages sent while the Redis server is down are lost.

SignalR automatically reconnects when the Redis server is available again.

Custom behavior for connection failures

Here's an example that shows how to handle Redis connection failure events.

C#

```
services.AddSignalR()
        .AddMessagePackProtocol()
        .AddStackExchangeRedis(o =>
        {
            o.ConnectionFactory = async writer =>
            {
                var config = new ConfigurationOptions
                {
                    AbortOnConnectFail = false
                };
                config.EndPoints.Add(IPAddress.Loopback, 0);
                config.SetDefaultPorts();
                var connection = await
ConnectionMultiplexer.ConnectAsync(config, writer);
                connection.ConnectionFailed += (_, e) =>
                {
                    Console.WriteLine("Connection to Redis failed.");
                };

                if (!connection.IsConnected)
                {
                    Console.WriteLine("Did not connect to Redis.");
                }

                return connection;
            };
        });
```

Redis Cluster

Redis Cluster utilizes multiple simultaneously active Redis servers to achieve high availability. When Redis Cluster is used as the backplane for SignalR, messages are delivered to all of the nodes of the cluster without code modifications to the app.

There's a tradeoff between the number of nodes in the cluster and the throughput of the backplane. Increasing the number of nodes increases the availability of the cluster but decreases the throughput because messages must be transmitted to all of the nodes in the cluster.

In the SignalR app, include all of the possible Redis nodes using either of the following approaches:

- List the nodes in the connection string delimited with commas.
- If using custom behavior for connection failures, add the nodes to ConfigurationOptions.Endpoints.

Next steps

For more information, see the following resources:

- ASP.NET Core SignalR production hosting and scaling
- Redis documentation
- StackExchange Redis documentation
- Azure Redis Cache documentation

Host ASP.NET Core SignalR in background services

Article • 06/18/2024

By Dave Pringle and Brady Gaster

This article provides guidance for:

- Hosting SignalR Hubs using a background worker process hosted with ASP.NET Core.
- Sending messages to connected clients from within a .NET Core BackgroundService.

View or download sample code (how to download)

Enable SignalR at app startup

Hosting ASP.NET Core SignalR Hubs in the context of a background worker process is identical to hosting a Hub in an ASP.NET Core web app. In `Program.cs`, calling `builder.Services.AddSignalR` adds the required services to the ASP.NET Core Dependency Injection (DI) layer to support SignalR. The `MapHub` method is called on the `WebApplication` `app` to connect the Hub endpoints in the ASP.NET Core request pipeline.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSignalR();
builder.Services.AddHostedService<Worker>();

var app = builder.Build();

app.MapHub<ClockHub>("/hubs/clock");

app.Run();
```

In the preceding example, the `ClockHub` class implements the `Hub<T>` class to create a strongly typed Hub. The `ClockHub` has been configured in `Program.cs` to respond to requests at the endpoint `/hubs/clock`.

For more information on strongly typed Hubs, see Use hubs in SignalR for ASP.NET Core.

> **Note**
>
> This functionality isn't limited to the Hub<T> class. Any class that inherits from Hub, such as DynamicHub, works.

C#

```
public class ClockHub : Hub<IClock>
{
    public async Task SendTimeToClients(DateTime dateTime)
    {
        await Clients.All.ShowTime(dateTime);
    }
}
```

The interface used by the strongly typed `ClockHub` is the `IClock` interface.

C#

```
public interface IClock
{
    Task ShowTime(DateTime currentTime);
}
```

Call a SignalR Hub from a background service

During startup, the `Worker` class, a `BackgroundService`, is enabled using `AddHostedService`.

C#

```
builder.Services.AddHostedService<Worker>();
```

Since SignalR is also enabled up during the startup phase, in which each Hub is attached to an individual endpoint in ASP.NET Core's HTTP request pipeline, each Hub is represented by an `IHubContext<T>` on the server. Using ASP.NET Core's DI features, other classes instantiated by the hosting layer, like `BackgroundService` classes, MVC Controller classes, or Razor page models, can get references to server-side Hubs by accepting instances of `IHubContext<ClockHub, IClock>` during construction.

C#

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;
    private readonly IHubContext<ClockHub, IClock> _clockHub;

    public Worker(ILogger<Worker> logger, IHubContext<ClockHub, IClock>
clockHub)
    {
        _logger = logger;
        _clockHub = clockHub;
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Worker running at: {Time}",
DateTime.Now);
            await _clockHub.Clients.All.ShowTime(DateTime.Now);
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

As the `ExecuteAsync` method is called iteratively in the background service, the server's current date and time are sent to the connected clients using the `ClockHub`.

React to SignalR events with background services

Like a Single Page App using the JavaScript client for SignalR, or a .NET desktop app using the ASP.NET Core SignalR .NET Client, a `BackgroundService` or `IHostedService` implementation can also be used to connect to SignalR Hubs and respond to events.

The `ClockHubClient` class implements both the `IClock` interface and the `IHostedService` interface. This way it can be enabled during startup to run continuously and respond to Hub events from the server.

C#

```
public partial class ClockHubClient : IClock, IHostedService
{
}
```

During initialization, the `ClockHubClient` creates an instance of a `HubConnection` and enables the `IClock.ShowTime` method as the handler for the Hub's `ShowTime` event.

C#

```
private readonly ILogger<ClockHubClient> _logger;
private HubConnection _connection;

public ClockHubClient(ILogger<ClockHubClient> logger)
{
    _logger = logger;

    _connection = new HubConnectionBuilder()
        .WithUrl(Strings.HubUrl)
        .Build();

    _connection.On<DateTime>(Strings.Events.TimeSent, ShowTime);
}

public Task ShowTime(DateTime currentTime)
{
    _logger.LogInformation("{CurrentTime}",
currentTime.ToShortTimeString());

    return Task.CompletedTask;
}
```

In the `IHostedService.StartAsync` implementation, the `HubConnection` is started asynchronously.

C#

```
public async Task StartAsync(CancellationToken cancellationToken)
{
    // Loop is here to wait until the server is running
    while (true)
    {
        try
        {
            await _connection.StartAsync(cancellationToken);

            break;
        }
        catch
        {
            await Task.Delay(1000, cancellationToken);
        }
    }
}
```

During the `IHostedService.StopAsync` method, the `HubConnection` is disposed of asynchronously.

C#

```
public async Task StopAsync(CancellationToken cancellationToken)
{
    await _connection.DisposeAsync();
}
```

Additional resources

- Get started
- Hubs
- Publish to Azure
- Strongly typed Hubs

ASP.NET Core SignalR configuration

Article • 09/18/2024

This article covers ASP.NET Core SignalR configuration.

For Blazor SignalR guidance, which adds to or supersedes the guidance in this article, see ASP.NET Core Blazor SignalR guidance.

JSON/MessagePack serialization options

ASP.NET Core SignalR supports two protocols for encoding messages: JSON and MessagePack. Each protocol has serialization configuration options.

JSON serialization can be configured on the server using the AddJsonProtocol extension method. `AddJsonProtocol` can be added after AddSignalR in `Startup.ConfigureServices`. The `AddJsonProtocol` method takes a delegate that receives an `options` object. The PayloadSerializerOptions property on that object is a `System.Text.Json` JsonSerializerOptions object that can be used to configure serialization of arguments and return values. For more information, see the System.Text.Json documentation.

For example, to configure the serializer to not change the casing of property names, rather than the default camel case names, use the following code in `Program.cs`:

C#

```
builder.Services.AddSignalR()
    .AddJsonProtocol(options => {
        options.PayloadSerializerOptions.PropertyNamingPolicy = null;
    });
```

In the .NET client, the same `AddJsonProtocol` extension method exists on HubConnectionBuilder. The `Microsoft.Extensions.DependencyInjection` namespace must be imported to resolve the extension method:

C#

```
// At the top of the file:
using Microsoft.Extensions.DependencyInjection;

// When constructing your connection:
var connection = new HubConnectionBuilder()
    .AddJsonProtocol(options => {
        options.PayloadSerializerOptions.PropertyNamingPolicy = null;
```

```
    })
    .Build();
```

> **Note**
>
> It's not possible to configure JSON serialization in the JavaScript client at this time.

Switch to Newtonsoft.Json

If you need features of `Newtonsoft.Json` that aren't supported in `System.Text.Json`, see Switch to Newtonsoft.Json.

MessagePack serialization options

MessagePack serialization can be configured by providing a delegate to the AddMessagePackProtocol call. See MessagePack in SignalR for more details.

> **Note**
>
> It's not possible to configure MessagePack serialization in the JavaScript client at this time.

Configure server options

The following table describes options for configuring SignalR hubs:

Expand table

Option	Default Value	Description
`ClientTimeoutInterval`	30 seconds	The server considers the client disconnected if it hasn't received a message (including keep-alive) in this interval. It could take longer than this timeout interval for the client to be marked disconnected due to how this is implemented. The recommended value is double the `KeepAliveInterval` value.

Option	Default Value	Description
`HandshakeTimeout`	15 seconds	If the client doesn't send an initial handshake message within this time interval, the connection is closed. This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification.
`KeepAliveInterval`	15 seconds	If the server hasn't sent a message within this interval, a ping message is sent automatically to keep the connection open. When changing `KeepAliveInterval`, change the `ServerTimeout` or `serverTimeoutInMilliseconds` setting on the client. The recommended `ServerTimeout` or `serverTimeoutInMilliseconds` value is double the `KeepAliveInterval` value.
`SupportedProtocols`	All installed protocols	Protocols supported by this hub. By default, all protocols registered on the server are allowed. Protocols can be removed from this list to disable specific protocols for individual hubs.
`EnableDetailedErrors`	`false`	If `true`, detailed exception messages are returned to clients when an exception is thrown in a Hub method. The default is `false` because these exception messages can contain sensitive information.
`StreamBufferCapacity`	`10`	The maximum number of items that can be buffered for client upload streams. If this limit is reached, the processing of invocations is blocked until the server processes stream items.
`MaximumReceiveMessageSize`	32 KB	Maximum size of a single incoming hub message. Increasing the value might increase the risk of Denial of service (DoS) attacks.
`MaximumParallelInvocationsPerClient`	1	The maximum number of hub methods that each client can call in parallel before queueing.

Option	Default Value	Description
`DisableImplicitFromServicesParameters`	`false`	Hub method arguments are resolved from DI if possible.

Options can be configured for all hubs by providing an options delegate to the `AddSignalR` call in `Program.cs`.

C#

```
builder.Services.AddSignalR(hubOptions =>
{
    hubOptions.EnableDetailedErrors = true;
    hubOptions.KeepAliveInterval = TimeSpan.FromMinutes(1);
});
```

Options for a single hub override the global options provided in `AddSignalR` and can be configured using AddHubOptions:

C#

```
builder.Services.AddSignalR().AddHubOptions<ChatHub>(options =>
{
    options.EnableDetailedErrors = true;
});
```

Advanced HTTP configuration options

Use `HttpConnectionDispatcherOptions` to configure advanced settings related to transports and memory buffer management. These options are configured by passing a delegate to MapHub in `Program.cs`.

C#

```
using Microsoft.AspNetCore.Http.Connections;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddSignalR();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
```

```
    app.UseHsts();
}

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

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();
app.MapHub<ChatHub>("/chathub", options =>
{
    options.Transports =
        HttpTransportType.WebSockets |
        HttpTransportType.LongPolling;
}
);
app.Run();
```

The following table describes options for configuring ASP.NET Core SignalR's advanced HTTP options:

Expand table

Option	Default Value	Description
`ApplicationMaxBufferSize`	64 KB	The maximum number of bytes received from the client that the server buffers before applying backpressure. Increasing this value allows the server to receive larger messages faster without applying backpressure, but can increase memory consumption.
`TransportMaxBufferSize`	64 KB	The maximum number of bytes sent by the app that the server buffers before observing backpressure. Increasing this value allows the server to buffer larger messages faster without awaiting backpressure, but can increase memory consumption.
`AuthorizationData`	Data automatically gathered from the `Authorize` attributes applied to the Hub class.	A list of IAuthorizeData objects used to determine if a client is authorized to connect to the hub.

Option	Default Value	Description
`Transports`	All Transports are enabled.	A bit flags enum of `HttpTransportType` values that can restrict the transports a client can use to connect.
`LongPolling`	See below.	Additional options specific to the Long Polling transport.
`WebSockets`	See below.	Additional options specific to the WebSockets transport.
`MinimumProtocolVersion`	0	Specify the minimum version of the negotiate protocol. This is used to limit clients to newer versions.
`CloseOnAuthenticationExpiration`	false	Set this option to enable authentication expiration tracking, which will close connections when a token expires.

The Long Polling transport has additional options that can be configured using the `LongPolling` property:

Expand table

Option	Default Value	Description
`PollTimeout`	90 seconds	The maximum amount of time the server waits for a message to send to the client before terminating a single poll request. Decreasing this value causes the client to issue new poll requests more frequently.

The WebSocket transport has additional options that can be configured using the `WebSockets` property:

Expand table

Option	Default Value	Description
`CloseTimeout`	5 seconds	After the server closes, if the client fails to close within this time interval, the connection is terminated.
`SubProtocolSelector`	`null`	A delegate that can be used to set the `Sec-WebSocket-Protocol` header to a custom value. The delegate receives the values requested by the client as input and is expected to return the desired value.

Configure client options

Client options can be configured on the `HubConnectionBuilder` type (available in the .NET and JavaScript clients). It's also available in the Java client, but the `HttpHubConnectionBuilder` subclass is what contains the builder configuration options, as well as on the `HubConnection` itself.

Configure logging

Logging is configured in the .NET Client using the `ConfigureLogging` method. Logging providers and filters can be registered in the same way as they are on the server. See the Logging in ASP.NET Core documentation for more information.

> **Note**
>
> In order to register Logging providers, you must install the necessary packages. See the Built-in logging providers section of the docs for a full list.

For example, to enable Console logging, install the `Microsoft.Extensions.Logging.Console` NuGet package. Call the `AddConsole` extension method:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub")
    .ConfigureLogging(logging => {
        logging.SetMinimumLevel(LogLevel.Information);
        logging.AddConsole();
    })
    .Build();
```

In the JavaScript client, a similar `configureLogging` method exists. Provide a `LogLevel` value indicating the minimum level of log messages to produce. Logs are written to the browser console window.

JavaScript

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

Instead of a `LogLevel` value, you can also provide a `string` value representing a log level name. This is useful when configuring SignalR logging in environments where you don't have access to the `LogLevel` constants.

JavaScript

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging("warn")
    .build();
```

The following table lists the available log levels. The value you provide to `configureLogging` sets the **minimum** log level that will be logged. Messages logged at this level, **or the levels listed after it in the table**, will be logged.

Expand table

String	LogLevel
`trace`	`LogLevel.Trace`
`debug`	`LogLevel.Debug`
`info` or `information`	`LogLevel.Information`
`warn` or `warning`	`LogLevel.Warning`
`error`	`LogLevel.Error`
`critical`	`LogLevel.Critical`
`none`	`LogLevel.None`

> **Note**
>
> To disable logging entirely, specify `signalR.LogLevel.None` in the `configureLogging` method.

For more information on logging, see the SignalR Diagnostics documentation.

The SignalR Java client uses the SLF4J library for logging. It's a high-level logging API that allows users of the library to choose their own specific logging implementation by bringing in a specific logging dependency. The following code snippet shows how to use `java.util.logging` with the SignalR Java client.

Gradle

```
implementation 'org.slf4j:slf4j-jdk14:1.7.25'
```

If you don't configure logging in your dependencies, SLF4J loads a default no-operation logger with the following warning message:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
```

This can safely be ignored.

Configure allowed transports

The transports used by SignalR can be configured in the `WithUrl` call (`withUrl` in JavaScript). A bitwise-OR of the values of `HttpTransportType` can be used to restrict the client to only use the specified transports. All transports are enabled by default.

For example, to disable the Server-Sent Events transport, but allow WebSockets and Long Polling connections:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", HttpTransportType.WebSockets |
HttpTransportType.LongPolling)
    .Build();
```

In the JavaScript client, transports are configured by setting the `transport` field on the options object provided to `withUrl`:

JavaScript

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", { transport: signalR.HttpTransportType.WebSockets |
signalR.HttpTransportType.LongPolling })
    .build();
```

In this version of the Java client WebSockets is the only available transport.

In the Java client, the transport is selected with the `withTransport` method on the `HttpHubConnectionBuilder`. The Java client defaults to using the WebSockets transport.

Java

```
HubConnection hubConnection =
HubConnectionBuilder.create("https://example.com/chathub")
    .withTransport(TransportEnum.WEBSOCKETS)
    .build();
```

> **Note**
>
> The SignalR Java client doesn't support transport fallback yet.

Configure bearer authentication

To provide authentication data along with SignalR requests, use the `AccessTokenProvider` option (`accessTokenFactory` in JavaScript) to specify a function that returns the desired access token. In the .NET Client, this access token is passed in as an HTTP "Bearer Authentication" token (Using the `Authorization` header with a type of `Bearer`). In the JavaScript client, the access token is used as a Bearer token, **except** in a few cases where browser APIs restrict the ability to apply headers (specifically, in Server-Sent Events and WebSockets requests). In these cases, the access token is provided as a query string value `access_token`.

In the .NET client, the `AccessTokenProvider` option can be specified using the options delegate in `WithUrl`:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.AccessTokenProvider = async () => {
            // Get and return the access token.
        };
    })
    .Build();
```

In the JavaScript client, the access token is configured by setting the `accessTokenFactory` field on the options object in `withUrl`:

JavaScript

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        accessTokenFactory: () => {
            // Get and return the access token.
            // This function can return a JavaScript Promise if asynchronous
            // logic is required to retrieve the access token.
        }
    })
    .build();
```

In the SignalR Java client, you can configure a bearer token to use for authentication by providing an access token factory to the HttpHubConnectionBuilder. Use withAccessTokenFactory to provide an RxJava Single<String>. With a call to Single.defer, you can write logic to produce access tokens for your client.

Java

```
HubConnection hubConnection =
HubConnectionBuilder.create("https://example.com/chathub")
    .withAccessTokenProvider(Single.defer(() -> {
        // Your logic here.
        return Single.just("An Access Token");
    })).build();
```

Configure timeout and keep-alive options

Additional options for configuring timeout and keep-alive behavior:

.NET

Expand table

Option	Default value	Description
`WithServerTimeout`	30 seconds (30,000 milliseconds)	Timeout for server activity and is set directly on HubConnectionBuilder. If the server hasn't sent a message in this interval, the client considers the server disconnected and triggers the `Closed` event (`onclose` in JavaScript). This value must be large enough for a ping message to be sent from the server **and** received by the client within the timeout interval. The recommended value is a number at least double the server's keep-alive

Option	Default value	Description
		interval (`WithKeepAliveInterval`) value to allow time for pings to arrive.
`HandshakeTimeout`	15 seconds	Timeout for initial server handshake and is available on the `HubConnection` object itself. If the server doesn't send a handshake response in this interval, the client cancels the handshake and triggers the `Closed` event (`onclose` in JavaScript). This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification .
`WithKeepAliveInterval`	15 seconds	Determines the interval at which the client sends ping messages and is set directly on HubConnectionBuilder. This setting allows the server to detect hard disconnects, such as when a client unplugs their computer from the network. Sending any message from the client resets the timer to the start of the interval. If the client hasn't sent a message in the `ClientTimeoutInterval` set on the server, the server considers the client disconnected.

In the .NET Client, timeout values are specified as `TimeSpan` values.

The following example shows values that are double the default values:

C#

```
var builder = new HubConnectionBuilder()
    .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
    .WithServerTimeout(TimeSpan.FromSeconds(60))
    .WithKeepAliveInterval(TimeSpan.FromSeconds(30))
    .Build();

builder.On<string, string>("ReceiveMessage", (user, message) => ...

await builder.StartAsync();
```

Configure stateful reconnect

SignalR stateful reconnect reduces the perceived downtime of clients that have a temporary disconnect in their network connection, such as when switching network connections or a short temporary loss in access.

Stateful reconnect achieves this by:

- Temporarily buffering data on the server and client.
- Acknowledging messages received (ACK-ing) by both the server and client.
- Recognizing when a connection is up and replaying messages that might have been sent while the connection was down.

Stateful reconnect is available in ASP.NET Core 8.0 and later.

Opt in to stateful reconnect at both the server hub endpoint and the client:

- Update the server hub endpoint configuration to enable the `AllowStatefulReconnects` option:

  C#

  ```
  app.MapHub<MyHub>("/hubName", options =>
  {
      options.AllowStatefulReconnects = true;
  });
  ```

  Optionally, the maximum buffer size in bytes allowed by the server can be set globally or for a specific hub with the `StatefulReconnectBufferSize` option:

  The `StatefulReconnectBufferSize` option set globally:

  C#

  ```
  builder.AddSignalR(o => o.StatefulReconnectBufferSize = 1000);
  ```

  The `StatefulReconnectBufferSize` option set for a specific hub:

  C#

  ```
  builder.AddSignalR().AddHubOptions<MyHub>(o =>
  o.StatefulReconnectBufferSize = 1000);
  ```

  The `StatefulReconnectBufferSize` option is optional with a default of 100,000 bytes.

- Update JavaScript or TypeScript client code to enable the `withStatefulReconnect` option:

  JavaScript

  ```
  const builder = new signalR.HubConnectionBuilder()
    .withUrl("/hubname")
    .withStatefulReconnect({ bufferSize: 1000 });  // Optional, defaults
  to 100,000
  const connection = builder.build();
  ```

  The `bufferSize` option is optional with a default of 100,000 bytes.

- Update .NET client code to enable the `WithStatefulReconnect` option:

  C#

  ```
    var builder = new HubConnectionBuilder()
        .WithUrl("<hub url>")
        .WithStatefulReconnect();
    builder.Services.Configure<HubConnectionOptions>(o =>
  o.StatefulReconnectBufferSize = 1000);
    var hubConnection = builder.Build();
  ```

  The `StatefulReconnectBufferSize` option is optional with a default of 100,000 bytes.

Configure additional options

Additional options can be configured in the `WithUrl` (`withUrl` in JavaScript) method on `HubConnectionBuilder` or on the various configuration APIs on the `HttpHubConnectionBuilder` in the Java client:

.NET

Expand table

.NET Option	Default value	Description
`AccessTokenProvider`	`null`	A function returning a string that is provided as a Bearer authentication token in HTTP requests.

.NET Option	Default value	Description
`SkipNegotiation`	`false`	Set this to `true` to skip the negotiation step. **Only supported when the WebSockets transport is the only enabled transport**. This setting can't be enabled when using the Azure SignalR Service.
`ClientCertificates`	Empty	A collection of TLS certificates to send to authenticate requests.
`Cookies`	Empty	A collection of HTTP cookies to send with every HTTP request.
`Credentials`	Empty	Credentials to send with every HTTP request.
`CloseTimeout`	5 seconds	WebSockets only. The maximum amount of time the client waits after closing for the server to acknowledge the close request. If the server doesn't acknowledge the close within this time, the client disconnects.
`Headers`	Empty	A Map of additional HTTP headers to send with every HTTP request.
`HttpMessageHandlerFactory`	`null`	A delegate that can be used to configure or replace the `HttpMessageHandler` used to send HTTP requests. Not used for WebSocket connections. This delegate must return a non-null value, and it receives the default value as a parameter. Either modify settings on that default value and return it, or return a new `HttpMessageHandler` instance. **When replacing the handler make sure to copy the settings you want to keep from the provided handler, otherwise, the configured options (such as Cookies and Headers) won't apply to the new handler.**
`Proxy`	`null`	An HTTP proxy to use when sending HTTP requests.
`UseDefaultCredentials`	`false`	Set this boolean to send the default credentials for HTTP and WebSockets requests. This enables the use of Windows authentication.
`WebSocketConfiguration`	`null`	A delegate that can be used to configure additional WebSocket options. Receives an instance of ClientWebSocketOptions that can be used to configure the options.
`ApplicationMaxBufferSize`	1 MB	The maximum number of bytes received from the server that the client buffers before applying

.NET Option	Default value	Description
		backpressure. Increasing this value allows the client to receive larger messages faster without applying backpressure, but can increase memory consumption.
`TransportMaxBufferSize`	1 MB	The maximum number of bytes sent by the user application that the client buffers before observing backpressure. Increasing this value allows the client to buffer larger messages faster without awaiting backpressure, but can increase memory consumption.

In the .NET Client, these options can be modified by the options delegate provided to `WithUrl`:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.Headers["Foo"] = "Bar";
        options.SkipNegotiation = true;
        options.Transports = HttpTransportType.WebSockets;
        options.Cookies.Add(new Cookie(/* ... */);
        options.ClientCertificates.Add(/* ... */);
    })
    .Build();
```

In the JavaScript Client, these options can be provided in a JavaScript object provided to `withUrl`:

JavaScript

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        // "Foo: Bar" will not be sent with WebSockets or Server-Sent Events
requests
        headers: { "Foo": "Bar" },
        transport: signalR.HttpTransportType.LongPolling
    })
    .build();
```

In the Java client, these options can be configured with the methods on the `HttpHubConnectionBuilder` returned from the `HubConnectionBuilder.create("HUB URL")`

Java

```
HubConnection hubConnection =
HubConnectionBuilder.create("https://example.com/chathub")
        .withHeader("Foo", "Bar")
        .shouldSkipNegotiate(true)
        .withHandshakeResponseTimeout(30*1000)
        .build();
```

Additional resources

- Get started with ASP.NET Core SignalR
- Use hubs in ASP.NET Core SignalR
- ASP.NET Core SignalR JavaScript client
- ASP.NET Core SignalR .NET Client
- Use MessagePack Hub Protocol in SignalR for ASP.NET Core
- ASP.NET Core SignalR supported platforms

Authentication and authorization in ASP.NET Core SignalR

Article • 06/18/2024

Authenticate users connecting to a SignalR hub

SignalR can be used with ASP.NET Core authentication to associate a user with each connection. In a hub, authentication data can be accessed from the HubConnectionContext.User property. Authentication allows the hub to call methods on all connections associated with a user. For more information, see Manage users and groups in SignalR. Multiple connections may be associated with a single user.

The following code is an example that uses SignalR and ASP.NET Core authentication:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using SignalRAuthenticationSample.Data;
using SignalRAuthenticationSample.Hubs;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
```

```
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.MapHub<ChatHub>("/chat");

app.Run();
```

> **Note**
>
> If a token expires during the lifetime of a connection, by default the connection continues to work. `LongPolling` and `ServerSentEvent` connections fail on subsequent requests if they don't send new access tokens. For connections to close when the authentication token expires, set CloseOnAuthenticationExpiration.

Cookie authentication

In a browser-based app, cookie authentication allows existing user credentials to automatically flow to SignalR connections. When using the browser client, no extra configuration is needed. If the user is logged in to an app, the SignalR connection automatically inherits this authentication.

Cookies are a browser-specific way to send access tokens, but non-browser clients can send them. When using the .NET Client, the `Cookies` property can be configured in the `.WithUrl` call to provide a cookie. However, using cookie authentication from the .NET client requires the app to provide an API to exchange authentication data for a cookie.

Bearer token authentication

The client can provide an access token instead of using a cookie. The server validates the token and uses it to identify the user. This validation is done only when the connection is established. During the life of the connection, the server doesn't automatically revalidate to check for token revocation.

In the JavaScript client, the token can be provided using the accessTokenFactory option.

TypeScript

```
// Connect, using the token we got.
this.connection = new signalR.HubConnectionBuilder()
    .withUrl("/hubs/chat", { accessTokenFactory: () => this.loginToken })
    .build();
```

In the .NET client, there's a similar AccessTokenProvider property that can be used to configure the token:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options =>
    {
        options.AccessTokenProvider = () => Task.FromResult(_myAccessToken);
    })
    .Build();
```

> **Note**
>
> The access token function provided is called before **every** HTTP request made by SignalR. If the token needs to be renewed in order to keep the connection active, do so from within this function and return the updated token. The token may need to be renewed so it doesn't expire during the connection.

In standard web APIs, bearer tokens are sent in an HTTP header. However, SignalR is unable to set these headers in browsers when using some transports. When using WebSockets and Server-Sent Events, the token is transmitted as a query string parameter.

Built-in JWT authentication

On the server, bearer token authentication is configured using the JWT Bearer middleware:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.SignalR;
using Microsoft.EntityFrameworkCore;
using SignalRAuthenticationSample.Data;
using SignalRAuthenticationSample.Hubs;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using SignalRAuthenticationSample;
```

```
var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddAuthentication(options =>
{
    // Identity made Cookie authentication the default.
    // However, we want JWT Bearer Auth to be the default.
    options.DefaultAuthenticateScheme =
JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
```

```
}).AddJwtBearer(options =>
  {
      // Configure the Authority to the expected value for
      // the authentication provider. This ensures the token
      // is appropriately validated.
      options.Authority = "Authority URL"; // TODO: Update URL

      // We have to hook the OnMessageReceived event in order to
      // allow the JWT authentication handler to read the access
      // token from the query string when a WebSocket or
      // Server-Sent Events request comes in.

      // Sending the access token in the query string is required when using
WebSockets or ServerSentEvents
      // due to a limitation in Browser APIs. We restrict it to only calls
to the
      // SignalR hub in this code.
      // See https://docs.microsoft.com/aspnet/core/signalr/security#access-
token-logging
      // for more information about security considerations when using
      // the query string to transmit the access token.
      options.Events = new JwtBearerEvents
      {
          OnMessageReceived = context =>
          {
              var accessToken = context.Request.Query["access_token"];

              // If the request is for our hub...
              var path = context.HttpContext.Request.Path;
              if (!string.IsNullOrEmpty(accessToken) &&
                  (path.StartsWithSegments("/hubs/chat")))
              {
                  // Read the token out of the query string
                  context.Token = accessToken;
              }
              return Task.CompletedTask;
          }
      };
  });
```

```
builder.Services.AddRazorPages();
builder.Services.AddSignalR();

// Change to use Name as the user identifier for SignalR
// WARNING: This requires that the source of your JWT token
// ensures that the Name claim is unique!
// If the Name claim isn't unique, users could receive messages
// intended for a different user!
builder.Services.AddSingleton<IUserIdProvider, NameUserIdProvider>();

// Change to use email as the user identifier for SignalR
// builder.Services.AddSingleton<IUserIdProvider, EmailBasedUserIdProvider>
();

// WARNING: use *either* the NameUserIdProvider *or* the
```

```
// EmailBasedUserIdProvider, but do not use both.

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.MapHub<ChatHub>("/chatHub");

app.Run();
```

> ⓘ Note
>
> The query string is used on browsers when connecting with WebSockets and Server-Sent Events due to browser API limitations. When using HTTPS, query string values are secured by the TLS connection. However, many servers log query string values. For more information, see Security considerations in ASP.NET Core SignalR. SignalR uses headers to transmit tokens in environments which support them (such as the .NET and Java clients).

Identity Server JWT authentication

When using Duende IdentityServer add a PostConfigureOptions<TOptions> service to the project:

C#

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.Extensions.Options;
public class ConfigureJwtBearerOptions :
IPostConfigureOptions<JwtBearerOptions>
{
```

```
    public void PostConfigure(string name, JwtBearerOptions options)
    {
        var originalOnMessageReceived = options.Events.OnMessageReceived;
        options.Events.OnMessageReceived = async context =>
        {
            await originalOnMessageReceived(context);

            if (string.IsNullOrEmpty(context.Token))
            {
                var accessToken = context.Request.Query["access_token"];
                var path = context.HttpContext.Request.Path;

                if (!string.IsNullOrEmpty(accessToken) &&
                    path.StartsWithSegments("/hubs"))
                {
                    context.Token = accessToken;
                }
            }
        };
    }
}
```

Register the service after adding services for authentication (AddAuthentication) and the authentication handler for Identity Server (AddIdentityServerJwt):

C#

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.Extensions.DependencyInjection.Extensions;
using SignalRAuthenticationSample.Hubs;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication()
    .AddIdentityServerJwt();
builder.Services.TryAddEnumerable(
    ServiceDescriptor.Singleton<IPostConfigureOptions<JwtBearerOptions>,
        ConfigureJwtBearerOptions>());

builder.Services.AddRazorPages();

var app = builder.Build();

// Code removed for brevity.
```

Cookies vs. bearer tokens

Cookies are specific to browsers. Sending them from other kinds of clients adds complexity compared to sending bearer tokens. Cookie authentication isn't recommended unless the app only needs to authenticate users from the browser client.

Bearer token authentication is the recommended approach when using clients other than the browser client.

Windows authentication

If Windows authentication is configured in the app, SignalR can use that identity to secure hubs. However, to send messages to individual users, add a custom User ID provider. The Windows authentication system doesn't provide the "Name Identifier" claim. SignalR uses the claim to determine the user name.

Add a new class that implements `IUserIdProvider` and retrieve one of the claims from the user to use as the identifier. For example, to use the "Name" claim (which is the Windows username in the form `[Domain]/[Username]`), create the following class:

C#

```
public class NameUserIdProvider : IUserIdProvider
{
    public string GetUserId(HubConnectionContext connection)
    {
        return connection.User?.Identity?.Name;
    }
}
```

Rather than `ClaimTypes.Name`, use any value from the `User`, such as the Windows SID identifier, etc.

> **Note**
>
> The value chosen must be unique among all the users in the system. Otherwise, a message intended for one user could end up going to a different user.

Register this component in `Program.cs`:

C#

```
using Microsoft.AspNetCore.Authentication.Negotiate;
using Microsoft.AspNetCore.SignalR;
using SignalRAuthenticationSample;

var builder = WebApplication.CreateBuilder(args);
var services = builder.Services;

services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
   .AddNegotiate();
```

```
services.AddAuthorization(options =>
{
    options.FallbackPolicy = options.DefaultPolicy;
});
services.AddRazorPages();

services.AddSignalR();
services.AddSingleton<IUserIdProvider, NameUserIdProvider>();

var app = builder.Build();

// Code removed for brevity.
```

In the .NET Client, Windows Authentication must be enabled by setting the UseDefaultCredentials property:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options =>
    {
        options.UseDefaultCredentials = true;
    })
    .Build();
```

Windows authentication is supported in Microsoft Edge, but not in all browsers. For example, in Chrome and Safari, attempting to use Windows authentication and WebSockets fails. When Windows authentication fails, the client attempts to fall back to other transports which might work.

Use claims to customize identity handling

An app that authenticates users can derive SignalR user IDs from user claims. To specify how SignalR creates user IDs, implement `IUserIdProvider` and register the implementation.

The sample code demonstrates how to use claims to select the user's email address as the identifying property.

> **Note**
>
> The value chosen must be unique among all the users in the system. Otherwise, a message intended for one user could end up going to a different user.

C#

```
public class EmailBasedUserIdProvider : IUserIdProvider
{
    public virtual string GetUserId(HubConnectionContext connection)
    {
        return connection.User?.FindFirst(ClaimTypes.Email)?.Value!;
    }
}
```

The account registration adds a claim with type `ClaimsTypes.Email` to the ASP.NET identity database.

C#

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl ??= Url.Content("~/");
    ExternalLogins = (await
_signInManager.GetExternalAuthenticationSchemesAsync())

.ToList();
    if (ModelState.IsValid)
    {
        var user = CreateUser();

        await _userStore.SetUserNameAsync(user, Input.Email,
CancellationToken.None);
        await _emailStore.SetEmailAsync(user, Input.Email,
CancellationToken.None);
        var result = await _userManager.CreateAsync(user, Input.Password);

        // Add the email claim and value for this user.
        await _userManager.AddClaimAsync(user, new Claim(ClaimTypes.Email,
Input.Email));

        // Remaining code removed for brevity.
```

Register this component in `Program.cs`:

C#

```
builder.Services.AddSingleton<IUserIdProvider, EmailBasedUserIdProvider>();
```

Authorize users to access hubs and hub methods

By default, all methods in a hub can be called by an unauthenticated user. To require authentication, apply the AuthorizeAttribute attribute to the hub:

C#

```
[Authorize]
public class ChatHub: Hub
{
}
```

The constructor arguments and properties of the `[Authorize]` attribute can be used to restrict access to only users matching specific authorization policies. For example, with the custom authorization policy called `MyAuthorizationPolicy`, only users matching that policy can access the hub using the following code:

C#

```
[Authorize("MyAuthorizationPolicy")]
public class ChatPolicyHub : Hub
{
    public override async Task OnConnectedAsync()
    {
        await Clients.All.SendAsync("ReceiveSystemMessage",
                                    $"{Context.UserIdentifier} joined.");
        await base.OnConnectedAsync();
    }
    // Code removed for brevity.
```

The `[Authorize]` attribute can be applied to individual hub methods. If the current user doesn't match the policy applied to the method, an error is returned to the caller:

C#

```
[Authorize]
public class ChatHub : Hub
{
    public async Task Send(string message)
    {
        // ... send a message to all users ...
    }

    [Authorize("Administrators")]
    public void BanUser(string userName)
    {
        // ... ban a user from the chat room (something only Administrators
can do) ...
    }
}
```

Use authorization handlers to customize hub method authorization

SignalR provides a custom resource to authorization handlers when a hub method requires authorization. The resource is an instance of HubInvocationContext. The `HubInvocationContext` includes the HubCallerContext, the name of the hub method being invoked, and the arguments to the hub method.

Consider the example of a chat room allowing multiple organization sign-in via Microsoft Entra ID. Anyone with a Microsoft account can sign in to chat, but only members of the owning organization should be able to ban users or view users' chat histories. Furthermore, we might want to restrict some functionality from specific users. Note how the `DomainRestrictedRequirement` serves as a custom IAuthorizationRequirement. Now that the `HubInvocationContext` resource parameter is being passed in, the internal logic can inspect the context in which the Hub is being called and make decisions on allowing the user to execute individual Hub methods:

C#

```
[Authorize]
public class ChatHub : Hub
{
    public void SendMessage(string message)
    {
    }

    [Authorize("DomainRestricted")]
    public void BanUser(string username)
    {
    }

    [Authorize("DomainRestricted")]
    public void ViewUserHistory(string username)
    {
    }
}
```

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.SignalR;

namespace SignalRAuthenticationSample;

public class DomainRestrictedRequirement :
    AuthorizationHandler<DomainRestrictedRequirement, HubInvocationContext>,
```

```
    IAuthorizationRequirement
{
    protected override Task
HandleRequirementAsync(AuthorizationHandlerContext context,
        DomainRestrictedRequirement requirement,
        HubInvocationContext resource)
    {
        if (context.User.Identity != null &&
          !string.IsNullOrEmpty(context.User.Identity.Name) &&
          IsUserAllowedToDoThis(resource.HubMethodName,
                                context.User.Identity.Name) &&
          context.User.Identity.Name.EndsWith("@microsoft.com"))
        {
                context.Succeed(requirement);

        }
        return Task.CompletedTask;
    }

    private bool IsUserAllowedToDoThis(string hubMethodName,
        string currentUsername)
    {
        return !(currentUsername.Equals("asdf42@microsoft.com") &&
            hubMethodName.Equals("banUser",
StringComparison.OrdinalIgnoreCase));
    }
}
```

In `Program.cs`, add the new policy, providing the custom `DomainRestrictedRequirement` requirement as a parameter to create the `DomainRestricted` policy:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using SignalRAuthenticationSample;
using SignalRAuthenticationSample.Data;
using SignalRAuthenticationSample.Hubs;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
var services = builder.Services;

services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
services.AddDatabaseDeveloperPageExceptionFilter();

services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

```
services.AddAuthorization(options =>
    {
        options.AddPolicy("DomainRestricted", policy =>
        {
            policy.Requirements.Add(new DomainRestrictedRequirement());
        });
    });

services.AddRazorPages();

var app = builder.Build();

// Code removed for brevity.
```

In the preceding example, the `DomainRestrictedRequirement` class is both an `IAuthorizationRequirement` and its own `AuthorizationHandler` for that requirement. It's acceptable to split these two components into separate classes to separate concerns. A benefit of the example's approach is there's no need to inject the `AuthorizationHandler` during startup, as the requirement and the handler are the same thing.

Additional resources

- Bearer Token Authentication in ASP.NET Core
- Resource-based Authorization
- View or download sample code (how to download)

Security considerations in ASP.NET Core SignalR

Article • 06/18/2024

By Andrew Stanton-Nurse

This article provides information on securing SignalR.

Cross-Origin Resource Sharing

Cross-Origin Resource Sharing (CORS) can be used to allow cross-origin SignalR connections in the browser. If JavaScript code is hosted on a different domain from the SignalR app, CORS middleware must be enabled to allow the JavaScript to connect to the SignalR app. Allow cross-origin requests only from domains you trust or control. For example:

- Your site is hosted on `http://www.example.com`
- Your SignalR app is hosted on `http://signalr.example.com`

CORS should be configured in the SignalR app to only allow the origin `www.example.com`.

For more information on configuring CORS, see Enable Cross-Origin Requests (CORS). SignalR **requires** the following CORS policies:

- Allow the specific expected origins. Allowing any origin is possible but is **not** secure or recommended.
- HTTP methods `GET` and `POST` must be allowed.
- Credentials must be allowed in order for cookie-based sticky sessions to work correctly. They must be enabled even when authentication isn't used.

However, in 5.0 we have provided an option in the TypeScript client to not use credentials. The option to not use credentials should only be used when you know 100% that credentials like Cookies are not needed in your app (cookies are used by azure app service when using multiple servers for sticky sessions).

For example, the following highlighted CORS policy allows a SignalR browser client hosted on `https://example.com` to access the SignalR app hosted on `https://signalr.example.com`:

C#

```
using SignalRChat.Hubs;

var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
                      policy =>
                      {
                          policy.WithOrigins("http://example.com");
                          policy.WithMethods("GET", "POST");
                          policy.AllowCredentials();
                      });
});

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddSignalR();

var app = builder.Build();

app.MapHub<ChatHub>("/chatHub");
```

In the previous example, the CORS policy is customized to allow specific origins, methods, and credentials. For more information on customizing CORS policies and middleware in ASP.NET Core, see CORS middleware: CORS with named policy and middleware.

WebSocket Origin Restriction

The protections provided by CORS don't apply to WebSockets. For origin restriction on WebSockets, read WebSockets origin restriction.

ConnectionId

Exposing `ConnectionId` can lead to malicious impersonation if the SignalR server or client version is ASP.NET Core 2.2 or earlier. If the SignalR server and client version are ASP.NET Core 3.0 or later, the `ConnectionToken` rather than the `ConnectionId` must be kept secret. The `ConnectionToken` is purposely not exposed in any API. It can be difficult to ensure that older SignalR clients aren't connecting to the server, so even if your SignalR server version is ASP.NET Core 3.0 or later, the `ConnectionId` shouldn't be exposed.

Access token logging

When using WebSockets or Server-Sent Events, the browser client sends the access token in the query string. Receiving the access token via query string is generally as secure as using the standard `Authorization` header. Always use HTTPS to ensure a secure end-to-end connection between the client and the server. Many web servers log the URL for each request, including the query string. Logging the URLs may log the access token. ASP.NET Core logs the URL for each request by default, which includes the query string. For example:

Output

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/chathub?
access_token=1234
```

If you have concerns about logging this data with your server logs, you can disable this logging entirely by configuring the `Microsoft.AspNetCore.Hosting` logger to the `Warning` level or above (these messages are written at `Info` level). For more information, see Apply log filter rules in code for more information. If you still want to log certain request information, you can write middleware to log the data that you require and filter out the `access_token` query string value (if present).

Exceptions

Exception messages are generally considered sensitive data that shouldn't be revealed to a client. By default, SignalR doesn't send the details of an exception thrown by a hub method to the client. Instead, the client receives a generic message indicating an error occurred. Exception message delivery to the client can be overridden (for example in development or test) with EnableDetailedErrors. Exception messages should not be exposed to the client in production apps.

Buffer management

SignalR uses per-connection buffers to manage incoming and outgoing messages. By default, SignalR limits these buffers to 32 KB. The largest message a client or server can send is 32 KB. The maximum memory consumed by a connection for messages is 32 KB. If your messages are always smaller than 32 KB, you can reduce the limit, which:

- Prevents a client from being able to send a larger message.
- The server will never need to allocate large buffers to accept messages.

If your messages are larger than 32 KB, you can increase the limit. Increasing this limit means:

- The client can cause the server to allocate large memory buffers.
- Server allocation of large buffers may reduce the number of concurrent connections.

There are limits for incoming and outgoing messages, both can be configured on the HttpConnectionDispatcherOptions object configured in `MapHub`:

- `ApplicationMaxBufferSize` represents the maximum number of bytes from the client that the server buffers. If the client attempts to send a message larger than this limit, the connection may be closed.
- `TransportMaxBufferSize` represents the maximum number of bytes the server can send. If the server attempts to send a message (including return values from hub methods) larger than this limit, an exception will be thrown.

Setting the limit to `0` disables the limit. Removing the limit allows a client to send a message of any size. Malicious clients sending large messages can cause excess memory to be allocated. Excess memory usage can significantly reduce the number of concurrent connections.

Use MessagePack Hub Protocol in SignalR for ASP.NET Core

Article • 06/18/2024

This article assumes the reader is familiar with the topics covered in Get started with ASP.NET Core SignalR.

What is MessagePack?

MessagePack is a fast and compact binary serialization format. It's useful when performance and bandwidth are a concern because it creates smaller messages than JSON. The binary messages are unreadable when looking at network traces and logs unless the bytes are passed through a MessagePack parser. SignalR has built-in support for the MessagePack format and provides APIs for the client and server to use.

Configure MessagePack on the server

To enable the MessagePack Hub Protocol on the server, install the `Microsoft.AspNetCore.SignalR.Protocols.MessagePack` package in your app. In the `Startup.ConfigureServices` method, add `AddMessagePackProtocol` to the `AddSignalR` call to enable MessagePack support on the server.

C#

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

> **Note**
>
> JSON is enabled by default. Adding MessagePack enables support for both JSON and MessagePack clients.

To customize how MessagePack formats data, `AddMessagePackProtocol` takes a delegate for configuring options. In that delegate, the `SerializerOptions` property is used to configure MessagePack serialization options. For more information on how the resolvers work, visit the MessagePack library at MessagePack-CSharp. Attributes can be used on the objects you want to serialize to define how they should be handled.

C#

```
services.AddSignalR()
    .AddMessagePackProtocol(options =>
    {
        options.SerializerOptions = MessagePackSerializerOptions.Standard
            .WithResolver(new CustomResolver())
            .WithSecurity(MessagePackSecurity.UntrustedData);
    });
```

> ⚠ **Warning**
>
> We strongly recommend reviewing CVE-2020-5234 and applying the recommended patches. For example, calling `.WithSecurity(MessagePackSecurity.UntrustedData)` when replacing the `SerializerOptions`.

Configure MessagePack on the client

> ⓘ **Note**
>
> JSON is enabled by default for the supported clients. Clients can only support a single protocol. Adding MessagePack support replaces any previously configured protocols.

.NET client

To enable MessagePack in the .NET Client, install the `Microsoft.AspNetCore.SignalR.Protocols.MessagePack` package and call `AddMessagePackProtocol` on `HubConnectionBuilder`.

C#

```
using Microsoft.AspNetCore.SignalR.Client;
using Microsoft.Extensions.DependencyInjection;

var hubConnection = new HubConnectionBuilder()
                        .WithUrl("/chathub")
                        .AddMessagePackProtocol()
                        .Build();
```

> **Note**
>
> This `AddMessagePackProtocol` call takes a delegate for configuring options just like the server.

JavaScript client

MessagePack support for the JavaScript client is provided by the @microsoft/signalr-protocol-msgpack npm package. Install the package by executing the following command in a command shell:

Bash

```
npm install @microsoft/signalr-protocol-msgpack
```

After installing the npm package, the module can be used directly via a JavaScript module loader or imported into the browser by referencing the following file:

node_modules\@microsoft\signalr-protocol-msgpack\dist\browser\signalr-protocol-msgpack.js

The following required javaScript files must be referenced in the order shown below:

HTML

```
<script src="~/lib/signalr/signalr.js"></script>
<script src="~/lib/signalr/signalr-protocol-msgpack.js"></script>
```

Adding `.withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())` to the `HubConnectionBuilder` configures the client to use the MessagePack protocol when connecting to a server.

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())
    .build();
```

At this time, there are no configuration options for the MessagePack protocol on the JavaScript client.

Java client

To enable MessagePack with Java, install the `com.microsoft.signalr.messagepack` package. When using Gradle, add the following line to the `dependencies` section of the *build.gradle* file:

Gradle

```
implementation 'com.microsoft.signalr.messagepack:signalr-messagepack:5.0.0'
```

When using Maven, add the following lines inside the `<dependencies>` element of the `pom.xml` file:

XML

```
<dependency>
    <groupId>com.microsoft.signalr.messagepack</groupId>
    <artifactId>signalr</artifactId>
    <version>5.0.0</version>
</dependency>
```

Call `withHubProtocol(new MessagePackHubProtocol())` on `HubConnectionBuilder`.

Java

```
HubConnection messagePackConnection = HubConnectionBuilder.create("YOUR HUB
URL HERE")
    .withHubProtocol(new MessagePackHubProtocol())
    .build();
```

MessagePack considerations

There are a few issues to be aware of when using the MessagePack Hub Protocol.

MessagePack is case-sensitive

The MessagePack protocol is case-sensitive. For example, consider the following C# class:

C#

```
public class ChatMessage
{
    public string Sender { get; }
```

```
    public string Message { get; }
}
```

When sending from the JavaScript client, you must use `PascalCased` property names, since the casing must match the C# class exactly. For example:

JavaScript

```
connection.invoke("SomeMethod", { Sender: "Sally", Message: "Hello!" });
```

Using `camelCased` names won't properly bind to the C# class. You can work around this by using the `Key` attribute to specify a different name for the MessagePack property. For more information, see the MessagePack-CSharp documentation.

DateTime.Kind is not preserved when serializing/deserializing

The MessagePack protocol doesn't provide a way to encode the `Kind` value of a `DateTime`. As a result, when deserializing a date, the MessagePack Hub Protocol will convert to the UTC format if the `DateTime.Kind` is `DateTimeKind.Local` otherwise it will not touch the time and pass it as is. If you're working with `DateTime` values, we recommend converting to UTC before sending them. Convert them from UTC to local time when you receive them.

MessagePack support in "ahead-of-time" compilation environment

The MessagePack-CSharp library used by the .NET client and server uses code generation to optimize serialization. As a result, it isn't supported by default on environments that use "ahead-of-time" compilation (such as Xamarin iOS or Unity). It's possible to use MessagePack in these environments by "pre-generating" the serializer/deserializer code. For more information, see the MessagePack-CSharp documentation. Once you have pre-generated the serializers, you can register them using the configuration delegate passed to `AddMessagePackProtocol`:

C#

```
services.AddSignalR()
    .AddMessagePackProtocol(options =>
    {
        StaticCompositeResolver.Instance.Register(
            MessagePack.Resolvers.GeneratedResolver.Instance,
```

```
            MessagePack.Resolvers.StandardResolver.Instance
        );
        options.SerializerOptions = MessagePackSerializerOptions.Standard
            .WithResolver(StaticCompositeResolver.Instance)
            .WithSecurity(MessagePackSecurity.UntrustedData);
    });
```

Type checks are more strict in MessagePack

The JSON Hub Protocol will perform type conversions during deserialization. For example, if the incoming object has a property value that is a number (`{ foo: 42 }`) but the property on the .NET class is of type `string`, the value will be converted. However, MessagePack doesn't perform this conversion and will throw an exception that can be seen in server-side logs (and in the console):

```
InvalidDataException: Error binding arguments. Make sure that the types of
the provided values match the types of the hub method being invoked.
```

For more information on this limitation, see GitHub issue aspnet/SignalR#2937.

Chars and Strings in Java

In the java client, `char` objects will be serialized as one-character `String` objects. This is in contrast with the C# and JavaScript client, which serialize them as `short` objects. The MessagePack spec itself does not define behavior for `char` objects, so it is up to the library author to determine how to serialize them. The difference in behavior between our clients is a result of the libraries we used for our implementations.

Additional resources

- ASP.NET Core SignalR .NET Client
- ASP.NET Core SignalR JavaScript client

Use streaming in ASP.NET Core SignalR

Article • 06/18/2024

By Brennan Conroy

ASP.NET Core SignalR supports streaming from client to server and from server to client. This is useful for scenarios where fragments of data arrive over time. When streaming, each fragment is sent to the client or server as soon as it becomes available, rather than waiting for all of the data to become available.

View or download sample code (how to download)

Set up a hub for streaming

A hub method automatically becomes a streaming hub method when it returns IAsyncEnumerable<T>, ChannelReader<T>, `Task<IAsyncEnumerable<T>>`, or `Task<ChannelReader<T>>`.

Server-to-client streaming

Streaming hub methods can return `IAsyncEnumerable<T>` in addition to `ChannelReader<T>`. The simplest way to return `IAsyncEnumerable<T>` is by making the hub method an async iterator method as the following sample demonstrates. Hub async iterator methods can accept a `CancellationToken` parameter that's triggered when the client unsubscribes from the stream. Async iterator methods avoid problems common with Channels, such as not returning the `ChannelReader` early enough or exiting the method without completing the ChannelWriter<T>.

> **Note**
>
> The following sample requires C# 8.0 or later.

C#

```
public class AsyncEnumerableHub : Hub
{
    public async IAsyncEnumerable<int> Counter(
        int count,
        int delay,
        [EnumeratorCancellation]
        CancellationToken cancellationToken)
```

```
    {
        for (var i = 0; i < count; i++)
        {
            // Check the cancellation token regularly so that the server
will stop
            // producing items if the client disconnects.
            cancellationToken.ThrowIfCancellationRequested();

            yield return i;

            // Use the cancellationToken in other APIs that accept
cancellation
            // tokens so the cancellation can flow down to them.
            await Task.Delay(delay, cancellationToken);
        }
    }
}
```

The following sample shows the basics of streaming data to the client using Channels. Whenever an object is written to the ChannelWriter<T>, the object is immediately sent to the client. At the end, the `ChannelWriter` is completed to tell the client the stream is closed.

> **Note**
>
> Write to the `ChannelWriter<T>` on a background thread and return the `ChannelReader` as soon as possible. Other hub invocations are blocked until a `ChannelReader` is returned.
>
> Wrap logic in a try ... catch statement. Complete the `Channel` in a finally block. If you want to flow an error, capture it inside the `catch` block and write it in the `finally` block.

C#

```
public ChannelReader<int> Counter(
    int count,
    int delay,
    CancellationToken cancellationToken)
{
    var channel = Channel.CreateUnbounded<int>();

    // We don't want to await WriteItemsAsync, otherwise we'd end up waiting
    // for all the items to be written before returning the channel back to
    // the client.
    _ = WriteItemsAsync(channel.Writer, count, delay, cancellationToken);

    return channel.Reader;
```

```
}

private async Task WriteItemsAsync(
    ChannelWriter<int> writer,
    int count,
    int delay,
    CancellationToken cancellationToken)
{
    Exception localException = null;
    try
    {
        for (var i = 0; i < count; i++)
        {
            await writer.WriteAsync(i, cancellationToken);

            // Use the cancellationToken in other APIs that accept
cancellation
            // tokens so the cancellation can flow down to them.
            await Task.Delay(delay, cancellationToken);
        }
    }
    catch (Exception ex)
    {
        localException = ex;
    }
    finally
    {
        writer.Complete(localException);
    }
}
```

Server-to-client streaming hub methods can accept a `CancellationToken` parameter that's triggered when the client unsubscribes from the stream. Use this token to stop the server operation and release any resources if the client disconnects before the end of the stream.

Client-to-server streaming

A hub method automatically becomes a client-to-server streaming hub method when it accepts one or more objects of type ChannelReader<T> or IAsyncEnumerable<T>. The following sample shows the basics of reading streaming data sent from the client. Whenever the client writes to the ChannelWriter<T>, the data is written into the `ChannelReader` on the server from which the hub method is reading.

C#

```
public async Task UploadStream(ChannelReader<string> stream)
{
    while (await stream.WaitToReadAsync())
```

```
    {
        while (stream.TryRead(out var item))
        {
            // do something with the stream item
            Console.WriteLine(item);
        }
    }
}
```

An IAsyncEnumerable<T> version of the method follows.

> ⓘ Note
>
> The following sample requires C# 8.0 or later.

C#

```
public async Task UploadStream(IAsyncEnumerable<string> stream)
{
    await foreach (var item in stream)
    {
        Console.WriteLine(item);
    }
}
```

.NET client

Server-to-client streaming

The `StreamAsync` and `StreamAsChannelAsync` methods on `HubConnection` are used to invoke server-to-client streaming methods. Pass the hub method name and arguments defined in the hub method to `StreamAsync` or `StreamAsChannelAsync`. The generic parameter on `StreamAsync<T>` and `StreamAsChannelAsync<T>` specifies the type of objects returned by the streaming method. An object of type `IAsyncEnumerable<T>` or `ChannelReader<T>` is returned from the stream invocation and represents the stream on the client.

A `StreamAsync` example that returns `IAsyncEnumerable<int>`:

C#

```
// Call "Cancel" on this CancellationTokenSource to send a cancellation
message to
// the server, which will trigger the corresponding token in the hub method.
```

```
var cancellationTokenSource = new CancellationTokenSource();
var stream = hubConnection.StreamAsync<int>(
    "Counter", 10, 500, cancellationTokenSource.Token);

await foreach (var count in stream)
{
    Console.WriteLine($"{count}");
}

Console.WriteLine("Streaming completed");
```

A corresponding `StreamAsChannelAsync` example that returns `ChannelReader<int>`:

C#

```
// Call "Cancel" on this CancellationTokenSource to send a cancellation
message to
// the server, which will trigger the corresponding token in the hub method.
var cancellationTokenSource = new CancellationTokenSource();
var channel = await hubConnection.StreamAsChannelAsync<int>(
    "Counter", 10, 500, cancellationTokenSource.Token);

// Wait asynchronously for data to become available
while (await channel.WaitToReadAsync())
{
    // Read all currently available data synchronously, before waiting for
more data
    while (channel.TryRead(out var count))
    {
        Console.WriteLine($"{count}");
    }
}

Console.WriteLine("Streaming completed");
```

In the previous code:

- The `StreamAsChannelAsync` method on `HubConnection` is used to invoke a server-to-client streaming method. Pass the hub method name and arguments defined in the hub method to `StreamAsChannelAsync`.
- The generic parameter on `StreamAsChannelAsync<T>` specifies the type of objects returned by the streaming method.
- A `ChannelReader<T>` is returned from the stream invocation and represents the stream on the client.

Client-to-server streaming

There are two ways to invoke a client-to-server streaming hub method from the .NET client. You can either pass in an `IAsyncEnumerable<T>` or a `ChannelReader` as an argument to `SendAsync`, `InvokeAsync`, or `StreamAsChannelAsync`, depending on the hub method invoked.

Whenever data is written to the `IAsyncEnumerable` or `ChannelWriter` object, the hub method on the server receives a new item with the data from the client.

If using an `IAsyncEnumerable` object, the stream ends after the method returning stream items exits.

> ⓘ **Note**
>
> The following sample requires C# 8.0 or later.

C#

```
async IAsyncEnumerable<string> clientStreamData()
{
    for (var i = 0; i < 5; i++)
    {
        var data = await FetchSomeData();
        yield return data;
    }
    //After the for loop has completed and the local function exits the
stream completion will be sent.
}

await connection.SendAsync("UploadStream", clientStreamData());
```

Or if you're using a `ChannelWriter`, you complete the channel with `channel.Writer.Complete()`:

C#

```
var channel = Channel.CreateBounded<string>(10);
await connection.SendAsync("UploadStream", channel.Reader);
await channel.Writer.WriteAsync("some data");
await channel.Writer.WriteAsync("some more data");
channel.Writer.Complete();
```

JavaScript client

Server-to-client streaming

JavaScript clients call server-to-client streaming methods on hubs with `connection.stream`. The `stream` method accepts two arguments:

- The name of the hub method. In the following example, the hub method name is `Counter`.
- Arguments defined in the hub method. In the following example, the arguments are a count for the number of stream items to receive and the delay between stream items.

`connection.stream` returns an `IStreamResult`, which contains a `subscribe` method. Pass an `IStreamSubscriber` to `subscribe` and set the `next`, `error`, and `complete` callbacks to receive notifications from the `stream` invocation.

JavaScript

```
connection.stream("Counter", 10, 500)
    .subscribe({
        next: (item) => {
            var li = document.createElement("li");
            li.textContent = item;
            document.getElementById("messagesList").appendChild(li);
        },
        complete: () => {
            var li = document.createElement("li");
            li.textContent = "Stream completed";
            document.getElementById("messagesList").appendChild(li);
        },
        error: (err) => {
            var li = document.createElement("li");
            li.textContent = err;
            document.getElementById("messagesList").appendChild(li);
        },
});
```

To end the stream from the client, call the `dispose` method on the `ISubscription` that's returned from the `subscribe` method. Calling this method causes cancellation of the `CancellationToken` parameter of the Hub method, if you provided one.

Client-to-server streaming

JavaScript clients call client-to-server streaming methods on hubs by passing in a `Subject` as an argument to `send`, `invoke`, or `stream`, depending on the hub method

invoked. The `Subject` is a class that looks like a `Subject`. For example in RxJS, you can use the Subject class from that library.

JavaScript

```
const subject = new signalR.Subject();
yield connection.send("UploadStream", subject);
var iteration = 0;
const intervalHandle = setInterval(() => {
    iteration++;
    subject.next(iteration.toString());
    if (iteration === 10) {
        clearInterval(intervalHandle);
        subject.complete();
    }
}, 500);
```

Calling `subject.next(item)` with an item writes the item to the stream, and the hub method receives the item on the server.

To end the stream, call `subject.complete()`.

Java client

Server-to-client streaming

The SignalR Java client uses the `stream` method to invoke streaming methods. `stream` accepts three or more arguments:

- The expected type of the stream items.
- The name of the hub method.
- Arguments defined in the hub method.

Java

```
hubConnection.stream(String.class, "ExampleStreamingHubMethod", "Arg1")
    .subscribe(
        (item) -> {/* Define your onNext handler here. */ },
        (error) -> {/* Define your onError handler here. */},
        () -> {/* Define your onCompleted handler here. */});
```

The `stream` method on `HubConnection` returns an Observable of the stream item type. The Observable type's `subscribe` method is where `onNext`, `onError` and `onCompleted` handlers are defined.

Client-to-server streaming

The SignalR Java client can call client-to-server streaming methods on hubs by passing in an Observable as an argument to `send`, `invoke`, or `stream`, depending on the hub method invoked.

Java

```
ReplaySubject<String> stream = ReplaySubject.create();
hubConnection.send("UploadStream", stream);
stream.onNext("FirstItem");
stream.onNext("SecondItem");
stream.onComplete();
```

Calling `stream.onNext(item)` with an item writes the item to the stream, and the hub method receives the item on the server.

To end the stream, call `stream.onComplete()`.

Additional resources

- Hubs
- .NET client
- JavaScript client
- Publish to Azure

Differences between ASP.NET SignalR and ASP.NET Core SignalR

Article • 06/18/2024

ASP.NET Core SignalR isn't compatible with clients or servers for ASP.NET SignalR. This article details features which have been removed or changed in ASP.NET Core SignalR.

How to identify the SignalR version

Expand table

	ASP.NET SignalR	ASP.NET Core SignalR
Server NuGet package	Microsoft.AspNet.SignalR	None. Included in the Microsoft.AspNetCore.App shared framework.
Client NuGet packages	Microsoft.AspNet.SignalR.Client Microsoft.AspNet.SignalR.JS	Microsoft.AspNetCore.SignalR.Client
JavaScript client npm package	signalr	@microsoft/signalr
Java client	GitHub Repository (deprecated)	Maven package com.microsoft.signalr
Server app type	ASP.NET (System.Web) or OWIN Self-Host	ASP.NET Core
Supported server platforms	.NET Framework 4.5 or later	.NET Core 3.0 or later

Feature differences

Automatic reconnects

In ASP.NET SignalR:

- By default, SignalR attempts to reconnect to the server if the connection is dropped.

In ASP.NET Core SignalR:

- Automatic reconnects are opt-in with both the .NET client and the JavaScript client:

C#

```
HubConnection connection = new HubConnectionBuilder()
    .WithUrl(new Uri("http://127.0.0.1:5000/chathub"))
    .WithAutomaticReconnect()
    .Build();
```

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect()
    .build();
```

Protocol support

ASP.NET Core SignalR supports JSON, as well as a new binary protocol based on MessagePack. Additionally, custom protocols can be created.

Transports

The Forever Frame transport isn't supported in ASP.NET Core SignalR.

Differences on the server

The ASP.NET Core SignalR server-side libraries are included in Microsoft.AspNetCore.App, which is used in the **ASP.NET Core Web Application** template for both Razor and MVC projects.

ASP.NET Core SignalR is an ASP.NET Core middleware. It must be configured by calling AddSignalR in `Startup.ConfigureServices`.

C#

```
services.AddSignalR()
```

To configure routing, map routes to hubs inside the UseEndpoints method call in the `Startup.Configure` method.

C#

```
app.UseRouting();

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

Sticky sessions

The scaleout model for ASP.NET SignalR allows clients to reconnect and send messages to any server in the farm. In ASP.NET Core SignalR, the client must interact with the same server for the duration of the connection. For scaleout using Redis, that means sticky sessions are required. For scaleout using Azure SignalR Service, sticky sessions aren't required because the service handles connections to clients.

Single hub per connection

In ASP.NET Core SignalR, the connection model has been simplified. Connections are made directly to a single hub, rather than a single connection being used to share access to multiple hubs.

Streaming

ASP.NET Core SignalR now supports streaming data from the hub to the client.

State

The ability to pass arbitrary state between clients and the hub (often called `HubState`) has been removed, as well as support for progress messages. There is no counterpart of hub proxies at the moment.

PersistentConnection removal

In ASP.NET Core SignalR, the PersistentConnection class has been removed.

GlobalHost

ASP.NET Core has dependency injection (DI) built into the framework. Services can use DI to access the HubContext. The `GlobalHost` object that is used in ASP.NET SignalR to get a `HubContext` doesn't exist in ASP.NET Core SignalR.

HubPipeline

ASP.NET Core SignalR doesn't have support for `HubPipeline` modules.

Differences on the client

TypeScript

The ASP.NET Core SignalR client is written in TypeScript. You can write in JavaScript or TypeScript when using the JavaScript client.

The JavaScript client is hosted at npm

In ASP.NET versions, the JavaScript client was obtained through a NuGet package in Visual Studio. In the ASP.NET Core versions, the @microsoft/signalr npm package contains the JavaScript libraries. This package isn't included in the **ASP.NET Core Web Application** template. Use npm to obtain and install the `@microsoft/signalr` npm package.

Console

```
npm init -y
npm install @microsoft/signalr
```

jQuery

The dependency on jQuery has been removed, however projects can still use jQuery.

Internet Explorer support

ASP.NET Core SignalR doesn't support Microsoft Internet Explorer, whereas ASP.NET SignalR supports Microsoft Internet Explorer 8 or later. For more information, see ASP.NET Core SignalR supported platforms.

JavaScript client method syntax

The JavaScript syntax has changed from the ASP.NET version of SignalR. Rather than using the `$connection` object, create a connection using the HubConnectionBuilder API.

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();
```

Use the on method to specify client methods that the hub can call.

JavaScript

```
connection.on("ReceiveMessage", (user, message) => {
    const msg = message.replace(/&/g, "&amp;").replace(/</g,
"&lt;").replace(/>/g, "&gt;");
    const encodedMsg = `${user} says ${msg}`;
    console.log(encodedMsg);
});
```

After creating the client method, start the hub connection. Chain a catch method to log or handle errors.

JavaScript

```
connection.start().catch(err => console.error(err));
```

Hub proxies

Hub proxies are no longer automatically generated. Instead, the method name is passed into the invoke API as a string.

.NET and other clients

The Microsoft.AspNetCore.SignalR.Client NuGet package contains the .NET client libraries for ASP.NET Core SignalR.

Use the HubConnectionBuilder to create and build an instance of a connection to a hub.

C#

```
connection = new HubConnectionBuilder()
    .WithUrl("url")
    .Build();
```

Scaleout differences

ASP.NET SignalR supports SQL Server and Redis. ASP.NET Core SignalR supports Azure SignalR Service and Redis.

ASP.NET

- SignalR scaleout with Azure Service Bus
- SignalR scaleout with Redis
- SignalR scaleout with SQL Server

ASP.NET Core

- Azure SignalR Service
- Redis Backplane

Additional resources

- Hubs
- JavaScript client
- .NET client
- Supported platforms

WebSockets support in ASP.NET Core

Article • 09/17/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

This article explains how to get started with WebSockets in ASP.NET Core. WebSocket (RFC 6455) is a protocol that enables two-way persistent communication channels over TCP connections. It's used in apps that benefit from fast, real-time communication, such as chat, dashboard, and game apps.

View or download sample code (how to download, how to run).

Http/2 WebSockets support

Using WebSockets over HTTP/2 takes advantage of new features such as:

- Header compression.
- Multiplexing, which reduces the time and resources needed when making multiple requests to the server.

These supported features are available in Kestrel on all HTTP/2 enabled platforms. The version negotiation is automatic in browsers and Kestrel, so no new APIs are needed.

.NET 7 introduced WebSockets over HTTP/2 support for Kestrel, the SignalR JavaScript client, and SignalR with Blazor WebAssembly.

> **Note**
>
> HTTP/2 WebSockets use CONNECT requests rather than GET, so your own routes and controllers may need updating. For more information, see **Add HTTP/2 WebSockets support for existing controllers** in this article.

Chrome and Edge have HTTP/2 WebSockets enabled by default, and you can enable it in FireFox on the `about:config` page with the `network.http.spdy.websockets` flag.

WebSockets were originally designed for HTTP/1.1 but have since been adapted to work over HTTP/2. (RFC 8441)

SignalR

ASP.NET Core SignalR is a library that simplifies adding real-time web functionality to apps. It uses WebSockets whenever possible.

For most applications, we recommend SignalR rather than raw WebSockets. SignalR:

- Provides transport fallback for environments where WebSockets isn't available.
- Provides a basic remote procedure call app model.
- Has no significant performance disadvantage compared to using raw WebSockets in most scenarios.

WebSockets over HTTP/2 is supported for:

- ASP.NET Core SignalR JavaScript client
- ASP.NET Core SignalR with Blazor WebAssembly

For some apps, gRPC on .NET provides an alternative to WebSockets.

Prerequisites

- Any OS that supports ASP.NET Core:
  - Windows 7 / Windows Server 2008 or later
  - Linux
  - macOS
- If the app runs on Windows with IIS:
  - Windows 8 / Windows Server 2012 or later
  - IIS 8 / IIS 8 Express
  - WebSockets must be enabled. See the IIS/IIS Express support section.
- If the app runs on HTTP.sys:
  - Windows 8 / Windows Server 2012 or later
- For supported browsers, see Can I use.

Configure the middleware

Add the WebSockets middleware in `Program.cs`:

```
C#

app.UseWebSockets();
```

The following settings can be configured:

- KeepAliveInterval - How frequently to send "ping" frames to the client to ensure proxies keep the connection open. The default is two minutes.
- AllowedOrigins - A list of allowed Origin header values for WebSocket requests. By default, all origins are allowed. For more information, see WebSocket origin restriction in this article.

```
C#

var webSocketOptions = new WebSocketOptions
{
    KeepAliveInterval = TimeSpan.FromMinutes(2)
};

app.UseWebSockets(webSocketOptions);
```

Accept WebSocket requests

Somewhere later in the request life cycle (later in `Program.cs` or in an action method, for example) check if it's a WebSocket request and accept the WebSocket request.

The following example is from later in `Program.cs`:

```
C#

app.Use(async (context, next) =>
{
    if (context.Request.Path == "/ws")
    {
        if (context.WebSockets.IsWebSocketRequest)
        {
            using var webSocket = await
context.WebSockets.AcceptWebSocketAsync();
            await Echo(webSocket);
        }
        else
        {
            context.Response.StatusCode = StatusCodes.Status400BadRequest;
        }
    }
```

```
    else
    {
        await next(context);
    }

});
```

A WebSocket request could come in on any URL, but this sample code only accepts requests for `/ws`.

A similar approach can be taken in a controller method:

C#

```
public class WebSocketController : ControllerBase
{
    [Route("/ws")]
    public async Task Get()
    {
        if (HttpContext.WebSockets.IsWebSocketRequest)
        {
            using var webSocket = await
HttpContext.WebSockets.AcceptWebSocketAsync();
            await Echo(webSocket);
        }
        else
        {
            HttpContext.Response.StatusCode =
StatusCodes.Status400BadRequest;
        }
    }
```

When using a WebSocket, you **must** keep the middleware pipeline running for the duration of the connection. If you attempt to send or receive a WebSocket message after the middleware pipeline ends, you may get an exception like the following:

```
System.Net.WebSockets.WebSocketException (0x80004005): The remote party
closed the WebSocket connection without completing the close handshake. --->
System.ObjectDisposedException: Cannot write to the response body, the
response has completed.
Object name: 'HttpResponseStream'.
```

If you're using a background service to write data to a WebSocket, make sure you keep the middleware pipeline running. Do this by using a TaskCompletionSource<TResult>. Pass the `TaskCompletionSource` to your background service and have it call TrySetResult

when you finish with the WebSocket. Then `await` the Task property during the request, as shown in the following example:

C#

```
app.Run(async (context) =>
{
    using var webSocket = await context.WebSockets.AcceptWebSocketAsync();
    var socketFinishedTcs = new TaskCompletionSource<object>();

    BackgroundSocketProcessor.AddSocket(webSocket, socketFinishedTcs);

    await socketFinishedTcs.Task;
});
```

The WebSocket closed exception can also happen when returning too soon from an action method. When accepting a socket in an action method, wait for the code that uses the socket to complete before returning from the action method.

Never use `Task.Wait`, `Task.Result`, or similar blocking calls to wait for the socket to complete, as that can cause serious threading issues. Always use `await`.

Add HTTP/2 WebSockets support for existing controllers

.NET 7 introduced WebSockets over HTTP/2 support for Kestrel, the SignalR JavaScript client, and SignalR with Blazor WebAssembly. HTTP/2 WebSockets use CONNECT requests rather than GET. If you previously used `[HttpGet("/path")]` on your controller action method for Websocket requests, update it to use `[Route("/path")]` instead.

C#

```
public class WebSocketController : ControllerBase
{
    [Route("/ws")]
    public async Task Get()
    {
        if (HttpContext.WebSockets.IsWebSocketRequest)
        {
            using var webSocket = await
HttpContext.WebSockets.AcceptWebSocketAsync();
            await Echo(webSocket);
        }
        else
        {
            HttpContext.Response.StatusCode =
```

```
StatusCodes.Status400BadRequest;
        }
    }
```

Compression

> **⚠ Warning**
>
> Enabling compression over encrypted connections can make an app subject to CRIME/BREACH attacks. If sending sensitive information, avoid enabling compression or use `WebSocketMessageFlags.DisableCompression` when calling `WebSocket.SendAsync`. This applies to both sides of the WebSocket. Note that the WebSockets API in the browser doesn't have configuration for disabling compression per send.

If compression of messages over WebSockets is desired, then the accept code must specify that it allows compression as follows:

C#

```
using (var webSocket = await context.WebSockets.AcceptWebSocketAsync(
    new WebSocketAcceptContext { DangerousEnableCompression = true }))
{

}
```

`WebSocketAcceptContext.ServerMaxWindowBits` and `WebSocketAcceptContext.DisableServerContextTakeover` are advanced options that control how the compression works.

Compression is negotiated between the client and server when first establishing a connection. You can read more about the negotiation in the Compression Extensions for WebSocket RFC.

> **ⓘ Note**
>
> If the compression negotiation isn't accepted by either the server or client, the connection is still established. However, the connection doesn't use compression when sending and receiving messages.

Send and receive messages

The `AcceptWebSocketAsync` method upgrades the TCP connection to a WebSocket connection and provides a WebSocket object. Use the `WebSocket` object to send and receive messages.

The code shown earlier that accepts the WebSocket request passes the `WebSocket` object to an `Echo` method. The code receives a message and immediately sends back the same message. Messages are sent and received in a loop until the client closes the connection:

C#

```
private static async Task Echo(WebSocket webSocket)
{
    var buffer = new byte[1024 * 4];
    var receiveResult = await webSocket.ReceiveAsync(
        new ArraySegment<byte>(buffer), CancellationToken.None);

    while (!receiveResult.CloseStatus.HasValue)
    {
        await webSocket.SendAsync(
            new ArraySegment<byte>(buffer, 0, receiveResult.Count),
            receiveResult.MessageType,
            receiveResult.EndOfMessage,
            CancellationToken.None);

        receiveResult = await webSocket.ReceiveAsync(
            new ArraySegment<byte>(buffer), CancellationToken.None);
    }

    await webSocket.CloseAsync(
        receiveResult.CloseStatus.Value,
        receiveResult.CloseStatusDescription,
        CancellationToken.None);
}
```

When accepting the WebSocket connection before beginning the loop, the middleware pipeline ends. Upon closing the socket, the pipeline unwinds. That is, the request stops moving forward in the pipeline when the WebSocket is accepted. When the loop is finished and the socket is closed, the request proceeds back up the pipeline.

Handle client disconnects

The server isn't automatically informed when the client disconnects due to loss of connectivity. The server receives a disconnect message only if the client sends it, which

can't be done if the internet connection is lost. If you want to take some action when that happens, set a timeout after nothing is received from the client within a certain time window.

If the client isn't always sending messages and you don't want to time out just because the connection goes idle, have the client use a timer to send a ping message every X seconds. On the server, if a message hasn't arrived within 2*X seconds after the previous one, terminate the connection and report that the client disconnected. Wait for twice the expected time interval to leave extra time for network delays that might hold up the ping message.

WebSocket origin restriction

The protections provided by CORS don't apply to WebSockets. Browsers do **not**:

- Perform CORS pre-flight requests.
- Respect the restrictions specified in `Access-Control` headers when making WebSocket requests.

However, browsers do send the `Origin` header when issuing WebSocket requests. Applications should be configured to validate these headers to ensure that only WebSockets coming from the expected origins are allowed.

If you're hosting your server on "https://server.com" and hosting your client on "https://client.com", add "https://client.com" to the AllowedOrigins list for WebSockets to verify.

C#

```
var webSocketOptions = new WebSocketOptions
{
    KeepAliveInterval = TimeSpan.FromMinutes(2)
};

webSocketOptions.AllowedOrigins.Add("https://client.com");
webSocketOptions.AllowedOrigins.Add("https://www.client.com");

app.UseWebSockets(webSocketOptions);
```

> ⓘ **Note**
>
> The `Origin` header is controlled by the client and, like the `Referer` header, can be faked. Do **not** use these headers as an authentication mechanism.

IIS/IIS Express support

Windows Server 2012 or later and Windows 8 or later with IIS/IIS Express 8 or later has support for the WebSocket protocol, but not for WebSockets over HTTP/2.

> ⓘ **Note**
>
> WebSockets are always enabled when using IIS Express.

Enabling WebSockets on IIS

To enable support for the WebSocket protocol on Windows Server 2012 or later:

> ⓘ **Note**
>
> These steps are not required when using IIS Express

1. Use the **Add Roles and Features** wizard from the **Manage** menu or the link in **Server Manager**.
2. Select **Role-based or Feature-based Installation**. Select **Next**.
3. Select the appropriate server (the local server is selected by default). Select **Next**.
4. Expand **Web Server (IIS)** in the **Roles** tree, expand **Web Server**, and then expand **Application Development**.
5. Select **WebSocket Protocol**. Select **Next**.
6. If additional features aren't needed, select **Next**.
7. Select **Install**.
8. When the installation completes, select **Close** to exit the wizard.

To enable support for the WebSocket protocol on Windows 8 or later:

> ⓘ **Note**
>
> These steps are not required when using IIS Express

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open the following nodes: **Internet Information Services** > **World Wide Web Services** > **Application Development Features**.
3. Select the **WebSocket Protocol** feature. Select **OK**.

Disable WebSocket when using socket.io on Node.js

If using the WebSocket support in socket.io on Node.js, disable the default IIS WebSocket module using the `webSocket` element in *web.config* or *applicationHost.config*. If this step isn't performed, the IIS WebSocket module attempts to handle the WebSocket communication rather than Node.js and the app.

XML

```
<system.webServer>
  <webSocket enabled="false" />
</system.webServer>
```

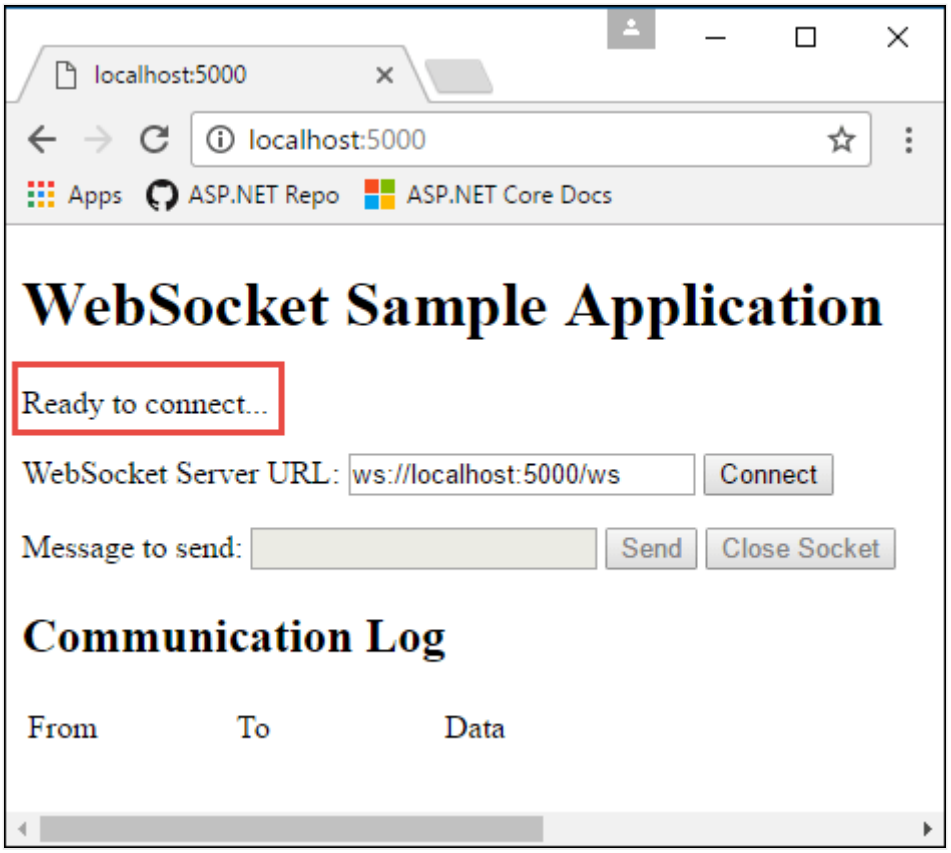
Sample app

The sample app that accompanies this article is an echo app. It has a webpage that makes WebSocket connections, and the server resends any messages it receives back to the client. The sample app supports WebSockets over HTTP/2 when using a targeted framework of .NET 7 or later.

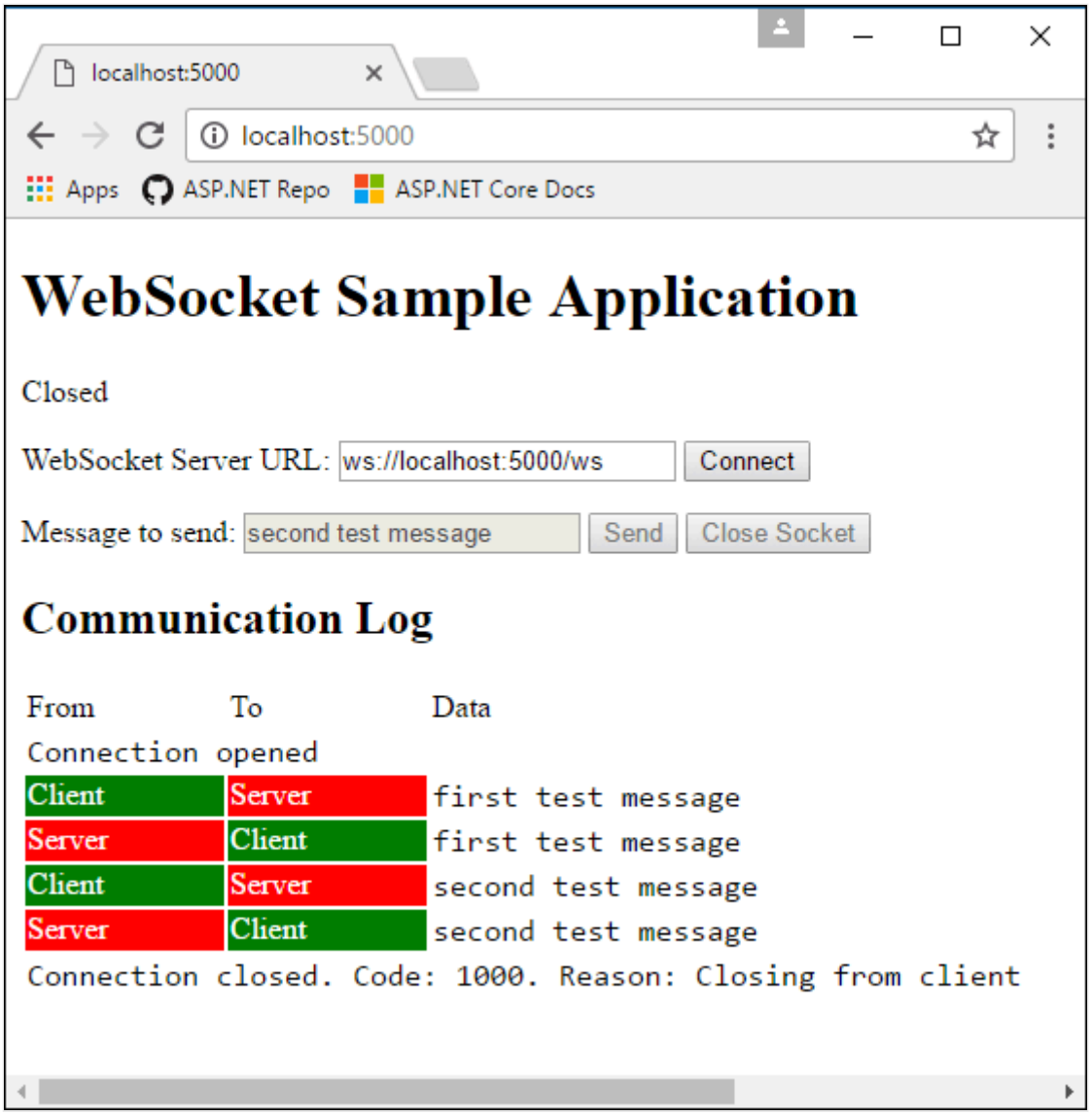
Run the app:

- To run app in Visual Studio: Open the sample project in Visual Studio, and press Ctrl+F5 to run without the debugger.
- To run the app in a command shell: Run the command dotnet run and navigate in a browser to `http://localhost:<port>`.

The webpage shows the connection status:



Select **Connect** to send a WebSocket request to the URL shown. Enter a test message and select **Send**. When done, select **Close Socket**. The **Communication Log** section reports each open, send, and close action as it happens.


localhost:5000
localhost:5000
Apps
ASP.NET Repo
ASP.NET Core Docs
WebSocket Sample Application
Closed
WebSocket Server URL:
ws://localhost:5000/ws
Connect
Message to send:
second test message
Send
Close Socket
Communication Log
From
To
Data
Connection opened
Client
Server
first test message
Server
Client
first test message
Client
Server
second test message
Server
Client
second test message
Connection closed. Code: 1000. Reason: Closing from client

Logging and diagnostics in ASP.NET Core SignalR

Article • 06/18/2024

By Andrew Stanton-Nurse

This article provides guidance for gathering diagnostics from your ASP.NET Core SignalR app to help troubleshoot issues.

Server-side logging

> ⚠ **Warning**
>
> Server-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

Since SignalR is part of ASP.NET Core, it uses the ASP.NET Core logging system. In the default configuration, SignalR logs very little information, but this can configured. See the documentation on ASP.NET Core logging for details on configuring ASP.NET Core logging.

SignalR uses two logger categories:

- `Microsoft.AspNetCore.SignalR`: For logs related to Hub Protocols, activating Hubs, invoking methods, and other Hub-related activities.
- `Microsoft.AspNetCore.Http.Connections`: For logs related to transports, such as WebSockets, Long Polling, Server-Sent Events, and low-level SignalR infrastructure.

To enable detailed logs from SignalR, configure both of the preceding prefixes to the `Debug` level in your `appsettings.json` file by adding the following items to the `LogLevel` sub-section in `Logging`:

JSON

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Debug",
            "System": "Information",
            "Microsoft": "Information",
            "Microsoft.AspNetCore.SignalR": "Debug",
            "Microsoft.AspNetCore.Http.Connections": "Debug"
```

```
        }
    }
}
```

You can also configure this in code in your `CreateWebHostBuilder` method:

C#

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureLogging(logging =>
        {
            logging.AddFilter("Microsoft.AspNetCore.SignalR",
LogLevel.Debug);
            logging.AddFilter("Microsoft.AspNetCore.Http.Connections",
LogLevel.Debug);
        })
        .UseStartup<Startup>();
```

If you aren't using JSON-based configuration, set the following configuration values in your configuration system:

- `Logging:LogLevel:Microsoft.AspNetCore.SignalR` = `Debug`
- `Logging:LogLevel:Microsoft.AspNetCore.Http.Connections` = `Debug`

Check the documentation for your configuration system to determine how to specify nested configuration values. For example, when using environment variables, two `_` characters are used instead of the `:` (for example, `Logging__LogLevel__Microsoft.AspNetCore.SignalR`).

We recommend using the `Debug` level when gathering more detailed diagnostics for your app. The `Trace` level produces very low-level diagnostics and is rarely needed to diagnose issues in your app.

Access server-side logs

How you access server-side logs depends on the environment in which you're running.

As a console app outside IIS

If you're running in a console app, the Console logger should be enabled by default. SignalR logs will appear in the console.

Within IIS Express from Visual Studio

Visual Studio displays the log output in the **Output** window. Select the **ASP.NET Core Web Server** drop down option.

Azure App Service

Enable the **Application Logging (Filesystem)** option in the **Diagnostics logs** section of the Azure App Service portal and configure the **Level** to `Verbose`. Logs should be available from the **Log streaming** service and in logs on the file system of the App Service. For more information, see Azure log streaming.

Other environments

If the app is deployed to another environment (for example, Docker, Kubernetes, or Windows Service), see Logging in .NET Core and ASP.NET Core for more information on how to configure logging providers suitable for the environment.

JavaScript client logging

> ⚠ **Warning**
>
> Client-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

When using the JavaScript client, you can configure logging options using the `configureLogging` method on `HubConnectionBuilder`:

JavaScript

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/my/hub/url")
    .configureLogging(signalR.LogLevel.Debug)
    .build();
```

To disable framework logging, specify `signalR.LogLevel.None` in the `configureLogging` method. Note that some logging is emitted directly by the browser and can't be disabled via setting the log level.

The following table shows log levels available to the JavaScript client. Setting the log level to one of these values enables logging at that level and all levels above it in the

table.

Expand table

Level	Description
`None`	No messages are logged.
`Critical`	Messages that indicate a failure in the entire app.
`Error`	Messages that indicate a failure in the current operation.
`Warning`	Messages that indicate a non-fatal problem.
`Information`	Informational messages.
`Debug`	Diagnostic messages useful for debugging.
`Trace`	Very detailed diagnostic messages designed for diagnosing specific issues.

Once you've configured the verbosity, the logs will be written to the Browser Console (or Standard Output in a NodeJS app).

If you want to send logs to a custom logging system, you can provide a JavaScript object implementing the `ILogger` interface. The only method that needs to be implemented is `log`, which takes the level of the event and the message associated with the event. For example:

TypeScript

```
import { ILogger, LogLevel, HubConnectionBuilder } from
"@microsoft/signalr";

export class MyLogger implements ILogger {
    log(logLevel: LogLevel, message: string) {
        // Use `message` and `logLevel` to record the log message to your
own system
    }
}

// later on, when configuring your connection...

let connection = new HubConnectionBuilder()
    .withUrl("/my/hub/url")
    .configureLogging(new MyLogger())
    .build();
```

.NET client logging

> ⚠ **Warning**
>
> Client-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

To get logs from the .NET client, you can use the `ConfigureLogging` method on `HubConnectionBuilder`. This works the same way as the `ConfigureLogging` method on `WebHostBuilder` and `HostBuilder`. You can configure the same logging providers you use in ASP.NET Core. However, you have to manually install and enable the NuGet packages for the individual logging providers.

To add .NET client logging to a Blazor WebAssembly app, see ASP.NET Core Blazor logging.

Console logging

In order to enable Console logging, add the Microsoft.Extensions.Logging.Console package. Then, use the `AddConsole` method to configure the console logger:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/my/hub/url")
    .ConfigureLogging(logging =>
    {
        // Log to the Console
        logging.AddConsole();

        // This will set ALL logging to Debug level
        logging.SetMinimumLevel(LogLevel.Debug);
    })
    .Build();
```

Debug output window logging

You can also configure logs to go to the **Output** window in Visual Studio. Install the Microsoft.Extensions.Logging.Debug package and use the `AddDebug` method:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/my/hub/url")
    .ConfigureLogging(logging =>
    {
```

```
        // Log to the Output Window
        logging.AddDebug();

        // This will set ALL logging to Debug level
        logging.SetMinimumLevel(LogLevel.Debug)
    })
    .Build();
```

Other logging providers

SignalR supports other logging providers such as Serilog, Seq, NLog, or any other logging system that integrates with `Microsoft.Extensions.Logging`. If your logging system provides an `ILoggerProvider`, you can register it with `AddProvider`:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/my/hub/url")
    .ConfigureLogging(logging =>
    {
        // Log to your custom provider
        logging.AddProvider(new MyCustomLoggingProvider());

        // This will set ALL logging to Debug level
        logging.SetMinimumLevel(LogLevel.Debug)
    })
    .Build();
```

Control verbosity

If you are logging from other places in your app, changing the default level to `Debug` may be too verbose. You can use a Filter to configure the logging level for SignalR logs. This can be done in code, in much the same way as on the server:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/my/hub/url")
    .ConfigureLogging(logging =>
    {
        // Register your providers

        // Set the default log level to Information, but to Debug for
SignalR-related loggers.
        logging.SetMinimumLevel(LogLevel.Information);
```

```
        logging.AddFilter("Microsoft.AspNetCore.SignalR", LogLevel.Debug);
        logging.AddFilter("Microsoft.AspNetCore.Http.Connections",
LogLevel.Debug);
    })
    .Build();
```

Network traces

> ⚠ **Warning**
>
> A network trace contains the full contents of every message sent by your app. **Never** post raw network traces from production apps to public forums like GitHub.

If you encounter an issue, a network trace can sometimes provide a lot of helpful information. This is particularly useful if you're going to file an issue on our issue tracker.

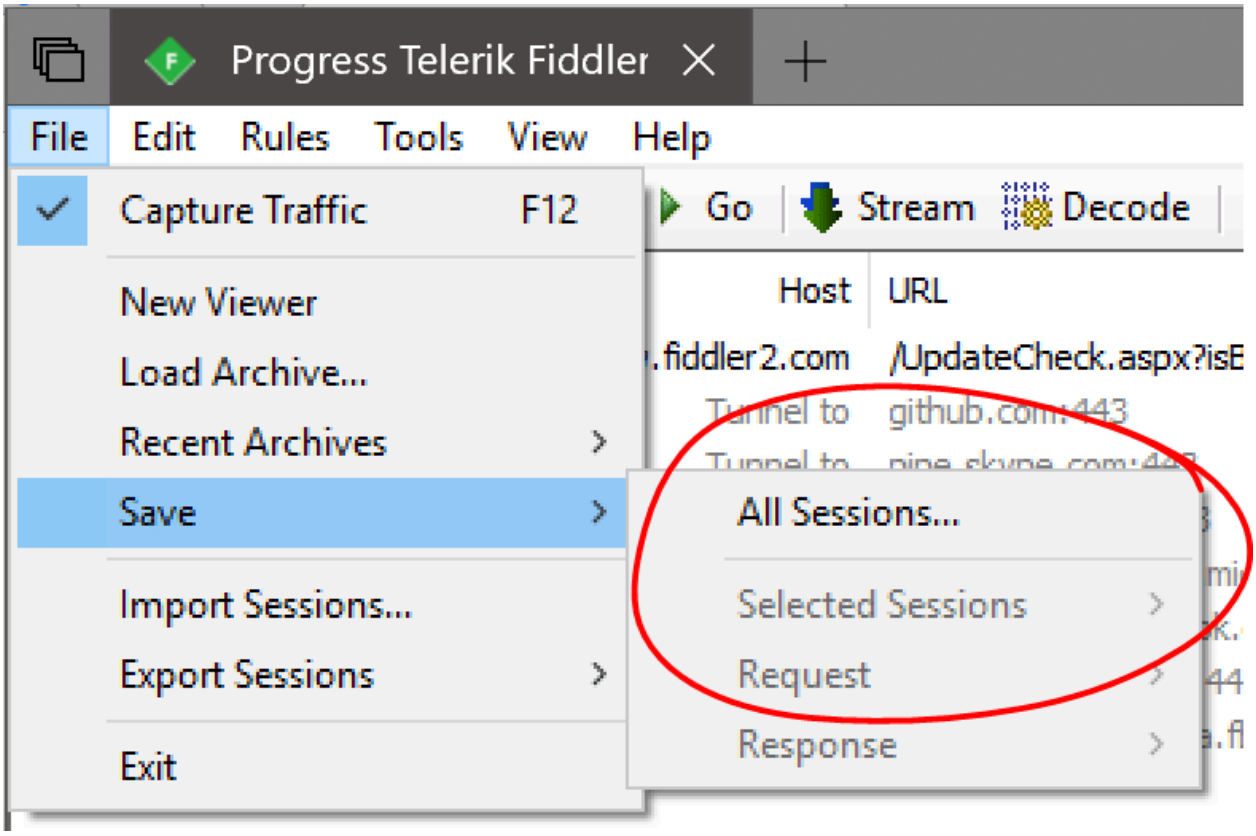
Collect a network trace with Fiddler (preferred option)

This method works for all apps.

Fiddler is a very powerful tool for collecting HTTP traces. Install it from telerik.com/fiddler, launch it, and then run your app and reproduce the issue. Fiddler is available for Windows, and there are beta versions for macOS and Linux.

If you connect using HTTPS, there are some extra steps to ensure Fiddler can decrypt the HTTPS traffic. For more details, see the Fiddler documentation.

Once you've collected the trace, you can export the trace by choosing **File** > **Save** > **All Sessions** from the menu bar.



Collect a network trace with tcpdump (macOS and Linux only)

This method works for all apps.

You can collect raw TCP traces using tcpdump by running the following command from a command shell. You may need to be `root` or prefix the command with `sudo` if you get a permissions error:

Console

```
tcpdump -i [interface] -w trace.pcap
```

Replace `[interface]` with the network interface you wish to capture on. Usually, this is something like `/dev/eth0` (for your standard Ethernet interface) or `/dev/lo0` (for localhost traffic). For more information, see the `tcpdump` man page on your host system.

Collect a network trace in the browser

This method only works for browser-based apps.

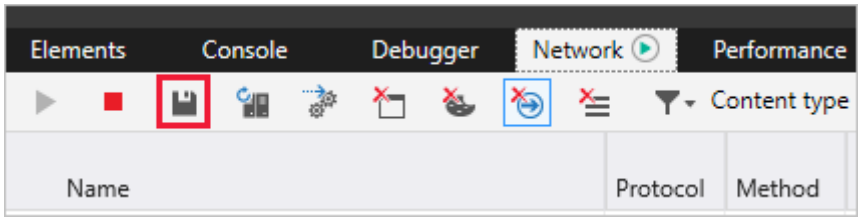
Most browser developer tools consoles have a "Network" tab that allows you to capture network activity between the browser and the server. However, these traces don't

include WebSocket and Server-Sent Event messages. If you are using those transports, using a tool like Fiddler or TcpDump (described below) is a better approach.

Microsoft Edge and Internet Explorer

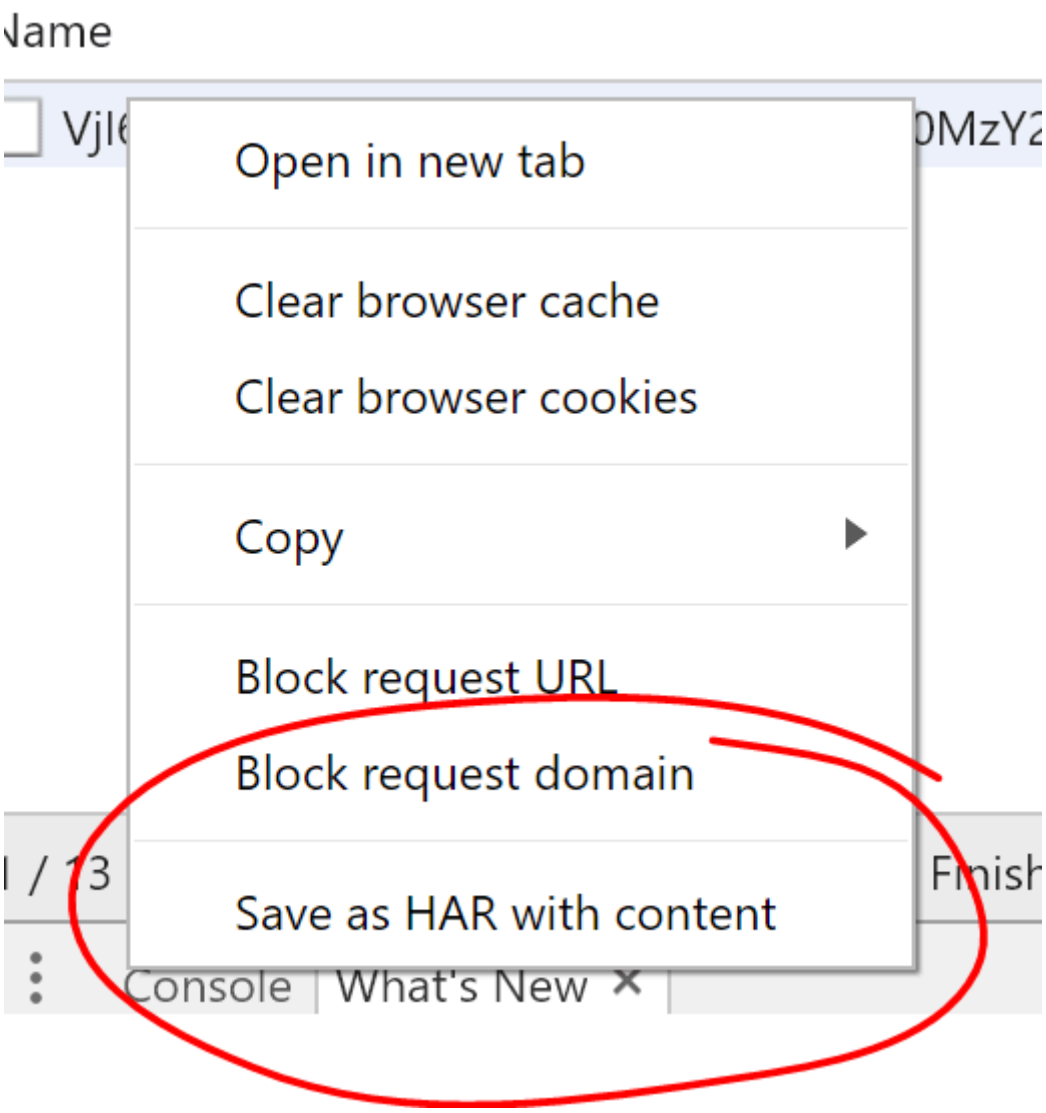
(The instructions are the same for both Edge and Internet Explorer)

1. Press F12 to open the Dev Tools
2. Click the Network Tab
3. Refresh the page (if needed) and reproduce the problem
4. Click the Save icon in the toolbar to export the trace as a "HAR" file:



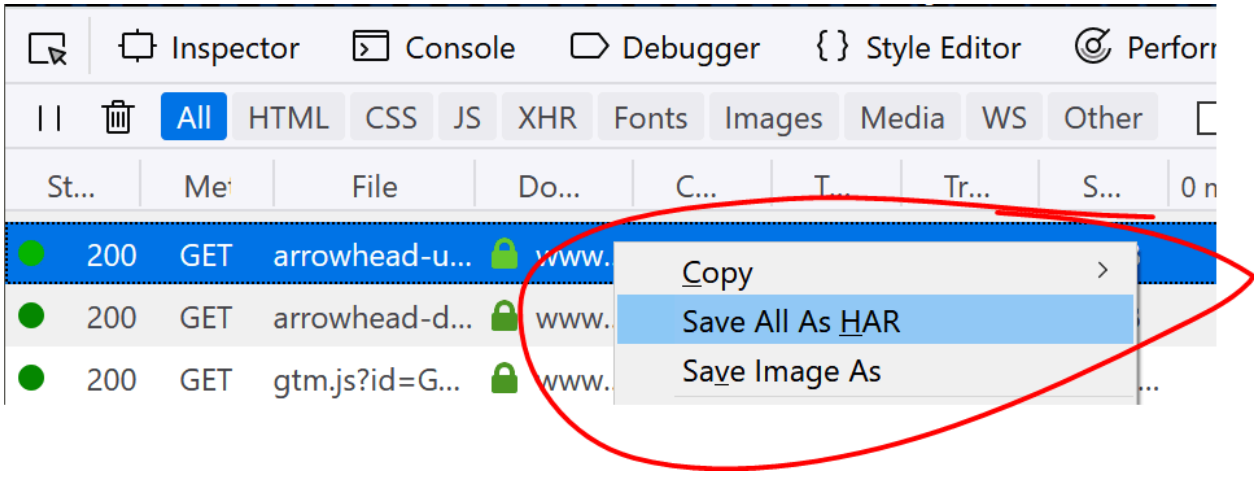
Google Chrome

1. Press F12 to open the Dev Tools
2. Click the Network Tab
3. Refresh the page (if needed) and reproduce the problem
4. Right click anywhere in the list of requests and choose "Save as HAR with content":



Mozilla Firefox

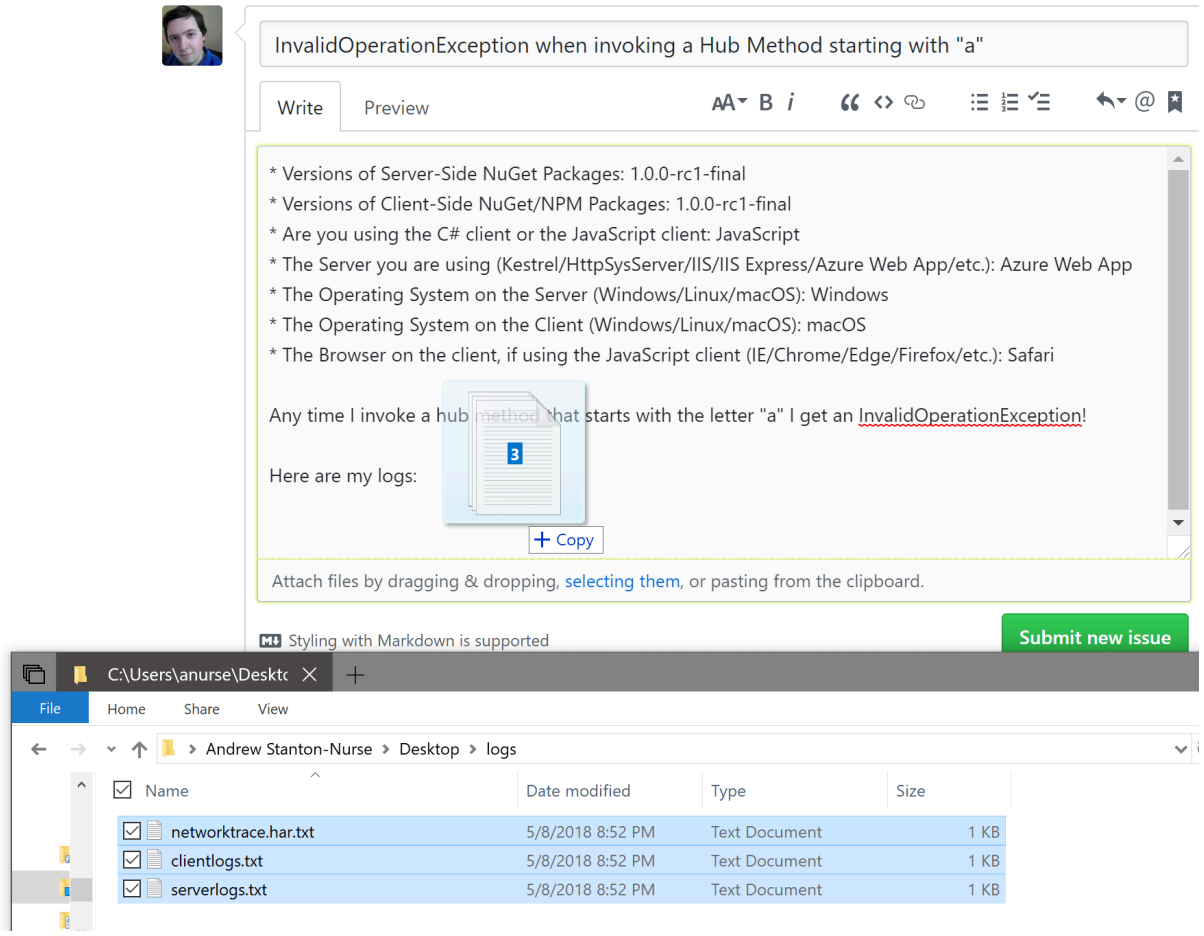
1. Press F12 to open the Dev Tools
2. Click the Network Tab
3. Refresh the page (if needed) and reproduce the problem
4. Right click anywhere in the list of requests and choose "Save All As HAR"



Attach diagnostics files to GitHub issues

You can attach Diagnostics files to GitHub issues by renaming them so they have a `.txt` extension and then dragging and dropping them on to the issue.

> **Note**
>
> Please don't paste the content of log files or network traces into a GitHub issue. These logs and traces can be quite large, and GitHub usually truncates them.



Metrics

Metrics is a representation of data measures over intervals of time. For example, requests per second. Metrics data allows observation of the state of an app at a high level. .NET gRPC metrics are emitted using EventCounter.

SignalR server metrics

SignalR server metrics are reported on the Microsoft.AspNetCore.Http.Connections event source.

Expand table

Name	Description
`connections-started`	Total connections started
`connections-stopped`	Total connections stopped
`connections-timed-out`	Total connections timed out
`current-connections`	Current connections
`connections-duration`	Average connection duration

Observe metrics

dotnet-counters is a performance monitoring tool for ad-hoc health monitoring and first-level performance investigation. Monitor a .NET app with `Microsoft.AspNetCore.Http.Connections` as the provider name. For example:

Console

```
> dotnet-counters monitor --process-id 37016
Microsoft.AspNetCore.Http.Connections

Press p to pause, r to resume, q to quit.
    Status: Running
[Microsoft.AspNetCore.Http.Connections]
    Average Connection Duration (ms)       16,040.56
    Current Connections                            1
    Total Connections Started                      8
    Total Connections Stopped                      7
    Total Connections Timed Out                    0
```

Additional resources

- ASP.NET Core SignalR configuration
- ASP.NET Core SignalR JavaScript client
- ASP.NET Core SignalR .NET Client

Troubleshoot connection errors

Article • 06/18/2024

This section provides help with errors that can occur when trying to establish a connection to an ASP.NET Core SignalR hub.

Response code 404

When using WebSockets and `skipNegotiation = true`

log

```
WebSocket connection to 'wss://xxx/HubName' failed: Error during WebSocket
handshake: Unexpected response code: 404
```

- When using multiple servers without sticky sessions, the connection can start on one server and then switch to another server. The other server is not aware of the previous connection.
- Verify the client is connecting to the correct endpoint. For example, the server is hosted at `http://127.0.0.1:5000/hub/myHub` and client is trying to connect to `http://127.0.0.1:5000/myHub`.
- If the connection uses the ID and takes too long to send a request to the server after the negotiate, the server:
  - Deletes the ID.
  - Returns a 404.

Response code 400 or 503

For the following error:

log

```
WebSocket connection to 'wss://xxx/HubName' failed: Error during WebSocket
handshake: Unexpected response code: 400

Error: Failed to start the connection: Error: There was an error with the
transport.
```

This error is usually caused by a client using only the WebSockets transport but the WebSocket protocol isn't enabled on the server.

Response code 307

When using WebSockets and `skipNegotiation = true`

log

```
WebSocket connection to 'ws://xxx/HubName' failed: Error during WebSocket
handshake: Unexpected response code: 307
```

This error can also happen during the negotiate request.

Common cause:

- App is configured to enforce HTTPS by calling `UseHttpsRedirection` in `Startup`, or enforces HTTPS via URL rewrite rule.

Possible solution:

- Change the URL on the client side from "http" to "https".
  `.withUrl("https://xxx/HubName")`

Response code 405

Http status code 405 - Method Not Allowed

- The app doesn't have CORS enabled

Response code 0

Http status code 0 - Usually a CORS issue, no status code is given

log

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the
remote resource at http://localhost:5000/default/negotiate?
negotiateVersion=1. (Reason: CORS header 'Access-Control-Allow-Origin'
missing).
```

- Add the expected origins to `.WithOrigins(...)`

log

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the
remote resource at http://localhost:5000/default/negotiate?
```

```
negotiateVersion=1. (Reason: expected 'true' in CORS header 'Access-Control-
Allow-Credentials').
```

- Add `.AllowCredentials()` to your CORS policy. Cannot use `.AllowAnyOrigin()` or `.WithOrigins("*")` with this option

Response code 413

Http status code 413 - Payload Too Large

This is often caused by having an access token that is over 4k.

- If using the Azure SignalR Service, reduce the token size by customizing the claims being sent through the Service with:

C#

```
.AddAzureSignalR(options =>
{
    options.ClaimsProvider = context => context.User.Claims;
});
```

Transient network failures

Transient network failures may close the SignalR connection. The server may interpret the closed connection as a graceful client disconnect. To get more info on why a client disconnected in those cases gather logs from the client and server.

Additional resources

- SignalR Hub Protocol

Overview for gRPC on .NET

Article • 07/31/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By James Newton-King

gRPC is a language agnostic, high-performance Remote Procedure Call (RPC) framework.

The main benefits of gRPC are:

- Modern, high-performance, lightweight RPC framework.
- Contract-first API development, using Protocol Buffers by default, allowing for language agnostic implementations.
- Tooling available for many languages to generate strongly-typed servers and clients.
- Supports client, server, and bi-directional streaming calls.
- Reduced network usage with Protobuf binary serialization.

These benefits make gRPC ideal for:

- Lightweight microservices where efficiency is critical.
- Polyglot systems where multiple languages are required for development.
- Point-to-point real-time services that need to handle streaming requests or responses.

C# Tooling support for `.proto` files

gRPC uses a contract-first approach to API development. Services and messages are defined in `.proto` files:

ProtoBuf

```
syntax = "proto3";
```

```
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

.NET types for services, clients, and messages are automatically generated by including `.proto` files in a project:

- Add a package reference to Grpc.Tools package.
- Add `.proto` files to the `<Protobuf>` item group.

XML

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" />
</ItemGroup>
```

For more information on gRPC tooling support, see gRPC services with C#.

gRPC services on ASP.NET Core

gRPC services can be hosted on ASP.NET Core. Services have full integration with ASP.NET Core features such as logging, dependency injection (DI), authentication, and authorization.

Add gRPC services to an ASP.NET Core app

gRPC requires the Grpc.AspNetCore package. For information on configuring gRPC in a .NET app, see Configure gRPC.

The gRPC service project template

The **ASP.NET Core gRPC Service** project template provides a starter service:

C#

```
public class GreeterService : Greeter.GreeterBase
{
```

```
    private readonly ILogger<GreeterService> _logger;

    public GreeterService(ILogger<GreeterService> logger)
    {
        _logger = logger;
    }

    public override Task<HelloReply> SayHello(HelloRequest request,
        ServerCallContext context)
    {
        _logger.LogInformation("Saying hello to {Name}", request.Name);
        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name
        });
    }
}
```

`GreeterService` inherits from the `GreeterBase` type, which is generated from the `Greeter` service in the `.proto` file. The service is made accessible to clients in `Program.cs`:

C#

```
app.MapGrpcService<GreeterService>();
```

To learn more about gRPC services on ASP.NET Core, see gRPC services with ASP.NET Core.

Call gRPC services with a .NET client

gRPC clients are concrete client types that are generated from .proto files. The concrete gRPC client has methods that translate to the gRPC service in the `.proto` file.

C#

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Greeter.GreeterClient(channel);

var response = await client.SayHelloAsync(
    new HelloRequest { Name = "World" });

Console.WriteLine(response.Message);
```

A gRPC client is created using a channel, which represents a long-lived connection to a gRPC service. A channel can be created using `GrpcChannel.ForAddress`.

For more information on creating clients, and calling different service methods, see Call gRPC services with the .NET client.

Additional resources

- gRPC services with C#
- gRPC services with ASP.NET Core
- Call gRPC services with the .NET client
- gRPC client factory integration in .NET
- Create a .NET Core gRPC client and server in ASP.NET Core

Tutorial: Create a gRPC client and server in ASP.NET Core

Article • 06/18/2024

This tutorial shows how to create a .NET Core gRPC client and an ASP.NET Core gRPC Server. At the end, you'll have a gRPC client that communicates with the gRPC Greeter service.

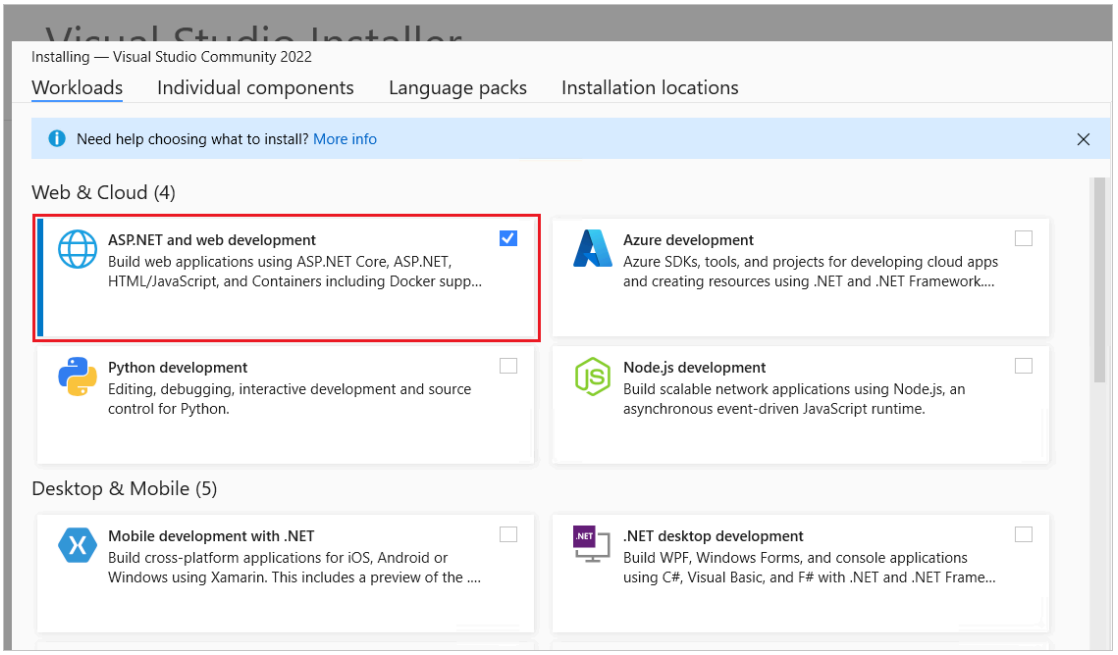
In this tutorial, you:

- ✓ Create a gRPC Server.
- ✓ Create a gRPC client.
- ✓ Test the gRPC client with the gRPC Greeter service.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Create a gRPC service

Visual Studio

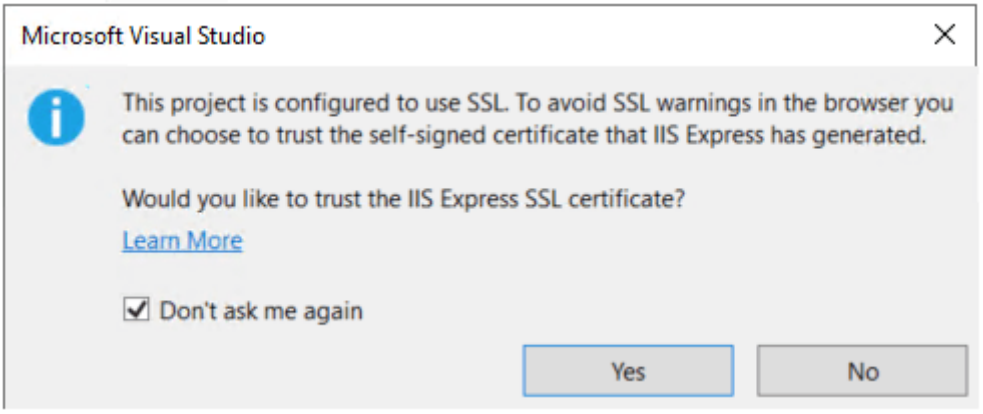
- Start Visual Studio 2022 and select **New Project**.
- In the **Create a new project** dialog, search for `gRPC`. Select **ASP.NET Core gRPC Service** and select **Next**.
- In the **Configure your new project** dialog, enter `GrpcGreeter` for **Project name**. It's important to name the project *GrpcGreeter* so the namespaces match when you copy and paste code.
- Select **Next**.
- In the **Additional information** dialog, select **.NET 8.0 (Long Term Support)** and then select **Create**.

Run the service

Visual Studio

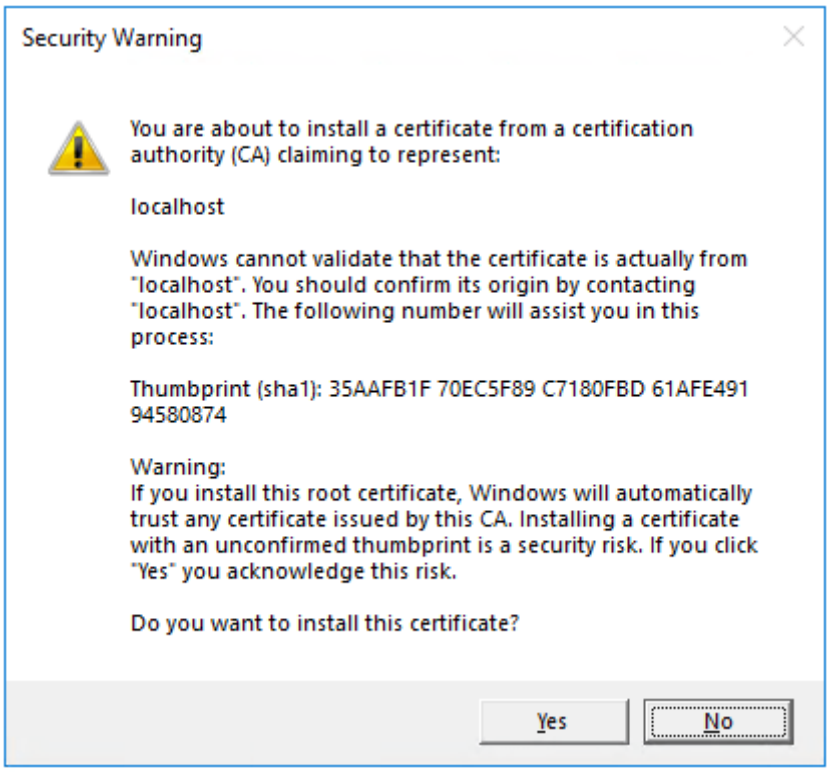
- Press Ctrl+F5 to run without the debugger.

  Visual Studio displays the following dialog when a project is not yet configured to use SSL:

  Select **Yes** if you trust the IIS Express SSL certificate.

  The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio:

- Starts Kestrel server.
- Launches a browser.
- Navigates to `http://localhost:port`, such as `http://localhost:7042`.
  - *port*: A randomly assigned port number for the app.
  - `localhost`: The standard hostname for the local computer. Localhost only serves web requests from the local computer.

The logs show the service listening on `https://localhost:<port>`, where `<port>` is the localhost port number randomly assigned when the project is created and set in `Properties/launchSettings.json`.

Console

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:<port>
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

> **Note**
>
> The gRPC template is configured to use Transport Layer Security (TLS). gRPC clients need to use HTTPS to call the server. The gRPC service localhost port number is randomly assigned when the project is created and set in the *Properties\launchSettings.json* file of the gRPC service project.

Examine the project files

GrpcGreeter project files:

- `Protos/greet.proto`: defines the `Greeter` gRPC and is used to generate the gRPC server assets. For more information, see Introduction to gRPC.
- `Services` folder: Contains the implementation of the `Greeter` service.
- `appSettings.json`: Contains configuration data such as the protocol used by Kestrel. For more information, see Configuration in ASP.NET Core.
- `Program.cs`, which contains:
  - The entry point for the gRPC service. For more information, see .NET Generic Host in ASP.NET Core.
  - Code that configures app behavior. For more information, see App startup.

Create the gRPC client in a .NET console app

Visual Studio

- Open a second instance of Visual Studio and select **New Project**.
- In the **Create a new project** dialog, select **Console App**, and select **Next**.
- In the **Project name** text box, enter **GrpcGreeterClient** and select **Next**.
- In the **Additional information** dialog, select **.NET 8.0 (Long Term Support)** and then select **Create**.

Add required NuGet packages

The gRPC client project requires the following NuGet packages:

- Grpc.Net.Client, which contains the .NET Core client.
- Google.Protobuf, which contains protobuf message APIs for C#.

- Grpc.Tools, which contain C# tooling support for protobuf files. The tooling package isn't required at runtime, so the dependency is marked with `PrivateAssets="All"`.

Visual Studio

Install the packages using either the Package Manager Console (PMC) or Manage NuGet Packages.

PMC option to install packages

- From Visual Studio, select **Tools** > **NuGet Package Manager** > **Package Manager Console**
- From the **Package Manager Console** window, run `cd GrpcGreeterClient` to change directories to the folder containing the `GrpcGreeterClient.csproj` files.
- Run the following commands:

PowerShell

```
Install-Package Grpc.Net.Client
Install-Package Google.Protobuf
Install-Package Grpc.Tools
```

Manage NuGet Packages option to install packages

- Right-click the project in **Solution Explorer** > **Manage NuGet Packages**.
- Select the **Browse** tab.
- Enter **Grpc.Net.Client** in the search box.
- Select the **Grpc.Net.Client** package from the **Browse** tab and select **Install**.
- Repeat for `Google.Protobuf` and `Grpc.Tools`.

Add greet.proto

- Create a *Protos* folder in the gRPC client project.
- Copy the *Protos\greet.proto* file from the gRPC Greeter service to the *Protos* folder in the gRPC client project.
- Update the namespace inside the `greet.proto` file to the project's namespace:

JSON

```
option csharp_namespace = "GrpcGreeterClient";
```

- Edit the `GrpcGreeterClient.csproj` project file:

Visual Studio

Right-click the project and select **Edit Project File**.

- Add an item group with a `<Protobuf>` element that refers to the *greet.proto* file:

XML

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
</ItemGroup>
```

Create the Greeter client

- Build the client project to create the types in the `GrpcGreeterClient` namespace.

> **Note**
>
> The `GrpcGreeterClient` types are generated automatically by the build process. The tooling package Grpc.Tools generates the following files based on the *greet.proto* file:
>
> - `GrpcGreeterClient\obj\Debug\[TARGET_FRAMEWORK]\Protos\Greet.cs`: The protocol buffer code which populates, serializes and retrieves the request and response message types.
> - `GrpcGreeterClient\obj\Debug\[TARGET_FRAMEWORK]\Protos\GreetGrpc.cs`: Contains the generated client classes.
>
> For more information on the C# assets automatically generated by Grpc.Tools, see gRPC services with C#: Generated C# assets.

- Update the gRPC client `Program.cs` file with the following code.

C#

```
using System.Threading.Tasks;
using Grpc.Net.Client;
using GrpcGreeterClient;

// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

- In the preceding highlighted code, replace the localhost port number `7042` with the `HTTPS` port number specified in `Properties/launchSettings.json` within the `GrpcGreeter` service project.

`Program.cs` contains the entry point and logic for the gRPC client.

The Greeter client is created by:

- Instantiating a `GrpcChannel` containing the information for creating the connection to the gRPC service.
- Using the `GrpcChannel` to construct the Greeter client:

C#

```
// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

The Greeter client calls the asynchronous `SayHello` method. The result of the `SayHello` call is displayed:

C#

```
// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
```

```
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

Test the gRPC client with the gRPC Greeter service

Update the `appsettings.Development.json` file by adding the following highlighted lines:

C#

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
      ,"Microsoft.AspNetCore.Hosting": "Information",
      "Microsoft.AspNetCore.Routing.EndpointMiddleware": "Information"
    }
  }
}
```

Visual Studio

- In the Greeter service, press `Ctrl+F5` to start the server without the debugger.
- In the `GrpcGreeterClient` project, press `Ctrl+F5` to start the client without the debugger.

The client sends a greeting to the service with a message containing its name, *GreeterClient*. The service sends the message "Hello GreeterClient" as a response. The "Hello GreeterClient" response is displayed in the command prompt:

Console

```
Greeting: Hello GreeterClient
Press any key to exit...
```

The gRPC service records the details of the successful call in the logs written to the command prompt:

Console

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:<port>
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path:
C:\GH\aspnet\docs\4\Docs\aspnetcore\tutorials\grpc\grpc-
start\sample\GrpcGreeter
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:
<port>/Greet.Greeter/SayHello application/grpc
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 78.32260000000001ms 200 application/grpc
```

> **Note**
>
> The code in this article requires the ASP.NET Core HTTPS development certificate to secure the gRPC service. If the .NET gRPC client fails with the message `The remote certificate is invalid according to the validation procedure.` or `The SSL connection could not be established.`, the development certificate isn't trusted. To fix this issue, see Call a gRPC service with an untrusted/invalid certificate.

Next steps

- View or download the completed sample code for this tutorial (how to download).
- Overview for gRPC on .NET
- gRPC services with C#
- Migrate gRPC from C-core to gRPC for .NET

gRPC services with C#

Article • 07/31/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

This document outlines the concepts needed to write gRPC apps in C#. The topics covered here apply to both C-core -based and ASP.NET Core-based gRPC apps.

proto file

gRPC uses a contract-first approach to API development. Protocol buffers (protobuf) are used as the Interface Definition Language (IDL) by default. The `.proto` file contains:

- The definition of the gRPC service.
- The messages sent between clients and servers.

For more information on the syntax of protobuf files, see Create Protobuf messages for .NET apps.

For example, consider the *greet.proto* file used in Get started with gRPC service:

- Defines a `Greeter` service.
- The `Greeter` service defines a `SayHello` call.
- `SayHello` sends a `HelloRequest` message and receives a `HelloReply` message:

ProtoBuf

```
syntax = "proto3";

option csharp_namespace = "GrpcGreeter";

package greet;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply);
```

```
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings.
message HelloReply {
  string message = 1;
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

Add a `.proto` file to a C# app

The `.proto` file is included in a project by adding it to the `<Protobuf>` item group:

XML

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
</ItemGroup>
```

By default, a `<Protobuf>` reference generates a concrete client and a service base class. The reference element's `GrpcServices` attribute can be used to limit C# asset generation. Valid `GrpcServices` options are:

- `Both` (default when not present)
- `Server`
- `Client`
- `None`

C# Tooling support for `.proto` files

The tooling package Grpc.Tools is required to generate the C# assets from `.proto` files. The generated assets (files):

- Are generated on an as-needed basis each time the project is built.
- Aren't added to the project or checked into source control.
- Are a build artifact contained in the *obj* directory.