**What is SignalR?**

ASP.NET Core SignalR is an open-source library that simplifies adding real-time web functionality to apps. Real-time web functionality enables server-side code to push content to clients instantly.

Good candidates for SignalR:

* Apps that require high frequency updates from the server. Examples are gaming, social networks, voting, auction, maps, and GPS apps.
* Dashboards and monitoring apps. Examples include company dashboards, instant sales updates, or travel alerts.
* Collaborative apps. Whiteboard apps and team meeting software are examples of collaborative apps.
* Apps that require notifications. Social networks, email, chat, games, travel alerts, and many other apps use notifications.

**Transports**

SignalR supports the following techniques for handling real-time communication (in order of graceful fallback):

* [WebSockets](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/websockets?view=aspnetcore-9.0)
* Server-Sent Events
* Long Polling

SignalR automatically chooses the best transport method that is within the capabilities of the server and client.

**Hubs**

SignalR uses *hubs* to communicate between clients and servers.

A hub is a high-level pipeline that allows a client and server to call methods on each other. SignalR handles the dispatching across machine boundaries automatically, allowing clients to call methods on the server and vice versa. You can pass strongly-typed parameters to methods, which enables model binding. SignalR provides two built-in hub protocols: a text protocol based on JSON and a binary protocol based on [MessagePack](https://msgpack.org/). MessagePack generally creates smaller messages compared to JSON. Older browsers must support [XHR level 2](https://caniuse.com/#feat=xhr2) to provide MessagePack protocol support.

**Create a SignalR hub**

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

In the SignalRChat project folder, create a Hubs folder.

In the Hubs folder, create the ChatHub class with the following code:

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);

}

}

}

**Configure SignalR**

The SignalR server must be configured to pass SignalR requests to SignalR. Add the following highlighted code to the Program.cs file.

C#Copy

using SignalRChat.Hubs;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddSignalR();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

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

app.Run();

**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#Copy

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);

**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:

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:

public async Task<IActionResult> Index()

{

await \_hubContext.Clients.All.SendAsync("Notify", $"Home page loaded at: {DateTime.Now}");

return View();

}

**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.

**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.

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.
* 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>();
* });
* }

**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#Copy

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);

}

}

**What is MessagePack?**

[MessagePack](https://msgpack.org/index.html) 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](https://www.json.org/). 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#Copy

services.AddSignalR()

.AddMessagePackProtocol();

**Use streaming in ASP.NET Core SignalR**

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.

**Set up a hub for streaming**

A hub method automatically becomes a streaming hub method when it returns [IAsyncEnumerable<T>](https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.iasyncenumerable-1), [ChannelReader<T>](https://learn.microsoft.com/en-us/dotnet/api/system.threading.channels.channelreader-1), Task<IAsyncEnumerable<T>>, or Task<ChannelReader<T>>.

**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>](https://learn.microsoft.com/en-us/dotnet/api/system.threading.channels.channelreader-1) or [IAsyncEnumerable<T>](https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.iasyncenumerable-1). The following sample shows the basics of reading streaming data sent from the client. Whenever the client writes to the [ChannelWriter<T>](https://learn.microsoft.com/en-us/dotnet/api/system.threading.channels.channelwriter-1), the data is written into the ChannelReader on the server from which the hub method is reading.

**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.