Use hubs in SignalR for ASP.NET Core

Article • 02/24/2023

By Rachel Appel and Kevin Griffin

The SignalR Hubs API enables connected clients to call methods on the server. The server defines methods that are called from the client and the client defines methods that are called from the server. 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:

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.
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

Property	Description
	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:

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:

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:

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

Method	Description
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 db-
Service)
    {
        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.

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 does not 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:

text

```
Microsoft.AspNetCore.SignalR.HubException: An unexpected error occurred invok-
```

```
ing '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

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more

.NET

ASP.NET Core feedback

The ASP.NET Core documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

information, see our contributor guide.