Use hubs in SignalR for ASP.NET Core

01/16/2020 • 7 minutes to read • +10

In this article

By Rachel Appel and Kevin Griffin

View or download sample code (how to download)

What is a SignalR hub

The SignalR Hubs API enables you to call methods on connected clients from the server. In the server code, you define methods that are called by client. In the client code, you define methods that are called from the server. SignalR takes care of everything behind the scenes that makes real-time client-to-server and server-to-client communications possible.

Configure SignalR hubs

The SignalR middleware requires some services, which are configured by calling `services.AddSignalR`.

C# Copy

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

When adding SignalR functionality to an ASP.NET Core app, setup SignalR routes by calling `endpoint.MapHub` in the `Startup.Configure` method's `app.UseEndpoints` callback.

C# Copy

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

Create and use hubs

Create a hub by declaring a class that inherits from `Hub`, and add public methods to it. Clients can call methods that are defined as `public`.

C# Copy

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

You can specify a return type and parameters, including complex types and arrays, as you would in any C# method. SignalR handles the serialization and deserialization of complex objects and arrays in your parameters and return values.

Note

Hubs are transient:

- Don't store state in a property on the hub class. Every hub method call is executed on a new hub instance.
- 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 has 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 is 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 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 is not associated with an HTTP request. For HTTP connections, you can use this method to get information such as HTTP headers and query strings.
`Abort`	Aborts the connection.

The Clients object

The `Hub` class has 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

Property	Description
`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
`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 allows you to supply the name and parameters of the client method to call.

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`.
- `SendMessageToGroups` sends a message to all clients in the `SignalR Users` group.

C# Copy

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

public Task SendMessageToCaller(string user, string message)
{
    return Clients.Caller.SendAsync("ReceiveMessage", user, message);
}

public Task SendMessageToGroup(string user, string message)
{
    return Clients.Group("SignalR Users").SendAsync("ReceiveMessage", user,
message);
}
```

Strongly typed hubs

A drawback of using `SendAsync` is that it relies on a magic 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` with Hub<T>. In the following example, the `ChatHub` client methods have been extracted out into an interface called `IChatClient`.

C# Copy

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

This interface can be used to refactor the preceding `ChatHub` example.

C# Copy

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

    public Task SendMessageToCaller(string user, string message)
    {
        return Clients.Caller.ReceiveMessage(user, message);
    }
}
```

Using `Hub<IChatClient>` enables compile-time checking of the client methods. This prevents issues caused by using magic 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`. Any methods defined on the interface can still be defined as asynchronous. In fact, each of these methods should return a `Task`. Since it's an interface, don't use the `async` keyword. For example:

C# Copy

```csharp
public interface IClient
{
    Task ClientMethod();
}
```

Note

The `Async` suffix isn't stripped from the method name. Unless your client method is defined with `.on('MyMethodAsync')`, you shouldn't use `MyMethodAsync` as a name.

Change the name of a hub method

By default, a server hub method name is the name of the .NET method. However, you can use the HubMethodName attribute to change this default and manually specify a name for the method. The client should use this name, instead of the .NET method name, when invoking the method.

C# Copy

```csharp
[HubMethodName("SendMessageToUser")]
public Task DirectMessage(string user, string message)
{
    return Clients.User(user).SendAsync("ReceiveMessage", user, message);
}
```

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# Copy

```
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 (by calling `connection.stop()`, for example), the `exception` parameter will be `null`. However, if the client is disconnected due to an error (such as a network failure), the `exception` parameter will contain an exception describing the failure.

C# Copy

```
public override async Task OnDisconnectedAsync(Exception exception)
{
    await Groups.RemoveFromGroupAsync(Context.ConnectionId, "SignalR 
Users");
    await base.OnDisconnectedAsync(exception);
}
```

⚠ **Warning**

Security warning: Exposing `ConnectionId` can lead to malicious impersonation if the SignalR server or client version is ASP.NET Core 2.2 or earlier.

Handle errors

Exceptions thrown in your hub methods are sent to the client that invoked the method. On the JavaScript client, the `invoke` method returns a JavaScript Promise . When the client receives an error with a handler attached to the promise using `catch`, it's invoked and passed as a JavaScript `Error` object.

JavaScript Copy

```
connection.invoke("SendMessage", user, message).catch(err => 
console.error(err));
```

If your Hub throws an exception, connections aren't closed. By default, SignalR returns a generic error message to the client. For example:

Copy

```
Microsoft.AspNetCore.SignalR.HubException: An unexpected error occurred in-
voking 'MethodName' 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. See the Security considerations article for more information on why exception details are suppressed.

If you have an exceptional condition you *do* want to propagate to the client, you can use the `HubException` class. If you throw a `HubException` from your hub method, SignalR **will** send the entire message to the client, unmodified.

C# Copy

```csharp
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.

Related resources

- Intro to ASP.NET Core SignalR
- JavaScript client
- Publish to Azure

Is this page helpful?

Yes No