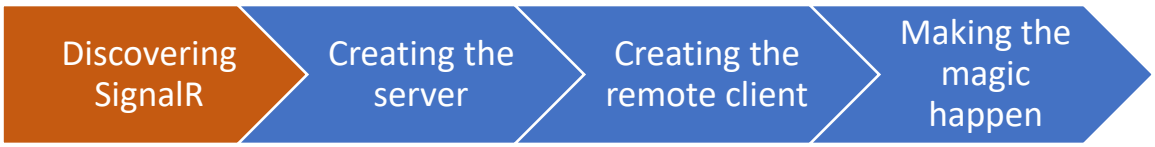
Dist. Sys. SignalR

The aim of this tutorial is to provide a practical introduction to use of SignalR by developing a simple application. SignalR allows us to make use of WebRTC, a realtime communications project which makes use of WebSockets to provide a fast and full-duplex connection. SignalR allows us to program at a much higher level of abstraction so we do not need to get our hands dirty with the actual low-level protocols or transfer of bytes.

We will be developing a basic chat server and considering some of the fundamentals of remote procedure calling over realtime connections, then looking at hosting in Kestrel for making connections over the LAN.

We will be following this process to start:



SignalR is a realtime communication library developed by Microsoft to facilitate Remote Procedure Calls between server and client. In order to provide maximum stability and speed SignalR keeps a bidirectional connection open between the two using the best available protocol (usually WebSockets). The only time the channel is closed is when the server or client disconnects or when Internet connection is lost.

This makes SignalR perfect for immediacy (e.g. chat, gaming, live web content, etc.) - when the server has something to send out to its clients it already has the connection made to do so, and the clients are listening for incoming messages. This is in contrast to other architectural styles such as request/response that only send messages in response to a request and therefore are not considered realtime.

The side effects of realtime connections are the limited number of possible concurrent connections that the server can manage. As each connection uses server resources there is a maximum number of connections that can be physically maintained per CPU and a maximum bandwidth throughput. This contrasts with request/response protocols, where connections are immediately closed after a response has been made.

This technology is different to remoting. Remoting uses proxies to obscure the underlying connection and allows you to use remote code as if it were local. SignalR, on the other hand, requires the use of handlers which manage the incoming message based on the method name being invoked.

Microsoft has developed their serverside SignalR libraries for ASP.NET, and you should already be quite familiar with this framework. A common use-case for SignalR is to facilitate communication through web browsers and, with this in mind, there are JavaScript libraries to handle all of the lower-level work for you on the client side too. Instead of this, we are going to create a desktop application that communicates with the server using Microsoft's SignalR C# client libraries. We will create a chatroom type of functionality.

SignalR Docs: https://learn.microsoft.com/en-gb/aspnet/core/signalr/introduction

In Visual Studio, create a new project and choose **ASP.NET Core Web App**. In this document the server project will be called SignalRLab. This will be your server application.

Choose **.NET 8 LTS** when given the option and deselect 'Configure for HTTPS'

Once the blank template has been created, add a new folder to your project called 'Hubs'.

Click on the folder and hit the key combination Ctrl+Shift+A. This will create a new class. Name it `ChatroomHub.cs`

Visual Studio will have created an empty class for you. Add this using directive to the top of the file:

```
using Microsoft.AspNetCore.SignalR;
```

Now we need to inherit from the base class Hub. This exists in the SignalR libraries we just added and contains fields and methods to manage our connections to clients. We can then start to add methods that clients can call remotely.

Modify the class definition to inherit from Hub:

```
public class ChatroomHub : Hub
```

Now we'll add a method for clients to call remotely. Add the method below:

```
public async Task BroadcastMessage(string username, string message)
{
    await Clients.All.SendAsync("GetMessage", username, message);
}
```

What do you notice about this method? First, it is an async method and it returns a Task. We've seen this in a previous workshop. This method receives a username and a message then sends the username and message to all connected clients. It is asynchronous, which means that the rest of our program can continue to function whilst our message is sent out to connected clients by the underlying framework.

Note the "GetMessage" string. This identifies the remote function that we will be calling on the client. When we implement the client we will need to make sure that we implement some functionality when we receive this call.

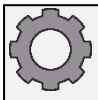
The class on its own is not much use, of course. It cannot be used unless we set up our server to function with SignalR. Thankfully that's not too difficult.

Open `Program.cs` and take a look at the default settings there. There are 3 areas here. The first is the creation of a builder.

The next two sections are the ones important to us. First, we have a section to configure the services – this will allow us to add services to our host environment. A service is an optional and reusable component that provides functionality for our app, this functionality is passed to our application code through dependency injection. There are lots of services already in the framework ready for us to configure and use. Today we're going to be using a SignalR service!

The next part is for configuration. This is where we can configure our app so that it works in the way we want it to. Again, there are lots of pre-made configurations, and often we only need to define a few properties to make them work correctly for us. The key thing to remember here is that ordering often matters because we are configuring the pipeline!

First, our Program file needs to be able to find the hub we just created.



Add the using directive to Program.cs so it knows where to look (inside our Hubs folder):

```
using SignalRLab.Hubs;
```

Now we'll add the service. Before the **`builder.Build()`** call, add this line:
`builder.Services.AddSignalR();`

Finally, we'll configure our app so is uses the SignalR service and knows where to route requests in middleware.

Under the `app.MapRazorPages()` call, add this line:
`app.MapHub<ChatroomHub>("/chatroom");`

This tells the routing system to map the path http://mydomain.com/chatroom to your ChatroomHub class. Now, if a request is made to that path, the framework knows to treat it as a SignalR connection because ChatroomHub inherits from **`Microsoft.AspNetCore.SignalR.Hub`**.

That's it! We have created a basic SignalR server hub and it's ready to be used.

The last thing we'll need to do is find out the port that our hub will be operating on. In the Solution Explorer window expand Properties and open `launchSettings.json`

You should see a JSON file with some settings.

Find the `"applicationUrl"` setting under iisSettings. It is likely to be something like 48400. Remember this number as we'll need it later. Where you see 48400 in the rest of the workshop, replace it with your port number.

IF YOU ARE DOING THIS ON A UNIVERSITY MACHINE AND LEFT CONFIGURE FOR HTTPS ON:

You may run into a trust issue where the computer won't trust self-signed certificates (this is very reasonable because it gives the impression of safety without a trusted intermediary proving the connection is secure).

This will cause connections to regularly fail. To fix this:

Open Program.cs and comment out the line:

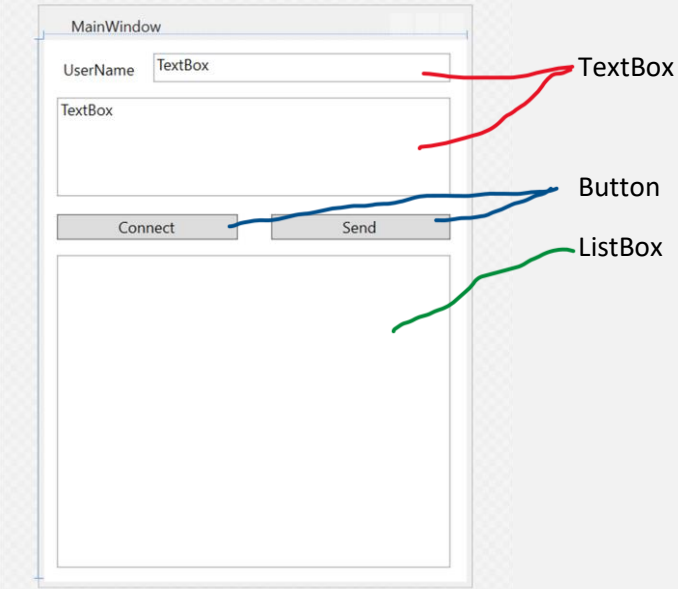
```
app.UseHttpsRedirection();
```

Ensure you are using the iisExpress `"applicationURL"` setting to get your port number rather than the `"sslPort"`.

Let's make a client to use our hub server.

Add a new project to your solution. Make this one a C# Windows Presentation Foundation (WPF) application called SignalRClient.

On the main form, add a few TextBoxes, some Connect and Send Buttons and a ListBox from the Toolbox so that your window looks something like this:



That's our UI set up. Now, let's get the SignalR client set up…

Right-click on your project in the Solution Explorer and click Manage NuGet Packages. Click Browse and search for: **`Microsoft.AspNetCore.SignalR.Client`**

Once found, click the package, choose **V 8.0.12** and click install. Wait for the package to install, it may take a minute or two.

Once installed, click on MainWindow.xaml in the Solution Explorer and press F7, this opens the interaction logic code for your WPF user interface.

Add this using directive to the top of the file:

```
using Microsoft.AspNetCore.SignalR.Client;
```

First, in the class, add a HubConnection. Your class should now look like this:

```
public partial class MainWindow : Window
{
    protected HubConnection connection;
    public MainWindow()
    {
        InitializeComponent();
    }
}
```

It makes sense to configure the connection when the window opens, so we'll set up the connection in the MainWindow constructor. Inside the constructor, add the lines:

```
connection = new HubConnectionBuilder()
   .WithUrl("http://localhost:48400/chatroom")
   .Build();
```

Remember to update the port number if your is different.

You will usually need to use `http` not `https` – we aren't using https because of self-signing issues on some machines but in a production environment we would want to use https as default.

Note: The path in our URL is /chatroom. See how this matches the route we specified in the server.

Now let's hook up our connect button. First we'll need a method that updates our ListBox.

Add this method to your `MainWindow` class:

```
private void GetMessage(string username, string message)
{
    this.Dispatcher.Invoke(() =>
    {
        var chat = $"{username}: {message}";
        MessagesListBox.Items.Add(chat);
    });
}
```

There are a few things of particular importance here.

1. The part highlighted in green. We need to invoke an action on the specific thread where the UI was created. Because we are acting asynchronously the UI needs to be updated via the dispatcher – so we are using a lambda expression to identify the function that we want to run. See Dispatcher: https://docs.microsoft.com/en-us/dotnet/api/system.windows.threading.dispatcher.invoke See Delegates and Lambda Expressions: https://docs.microsoft.com/en-us/dotnet/standard/delegates-lambdas

2. The part highlighted in yellow. This will be the name you gave your ListBox, which may be different. You can change the name of any of your elements in XAML (right click and press 'View Source') or in the Properties window of the designer. If yours is different you'll need to remember that for the rest of the lab and make changes where appropriate.

Next, we want to hook this functionality up so that when we are connected to the server, the "GetMessage" procedure is handled.

Inside the `MainWindow()` constructor method, underneath the new connection line add the line:

```
connection.On<string, string>("GetMessage",
    new Action<string, string>((username, message) =>
    GetMessage(username, message)));
```

What does this line actually do? Let's break it down:

`connection.On<string, string>("GetMessage",`

- This says when the connection receives a message from the hub with two strings as parameters and the method name "GetMessage" …

`new Action<string, string>((username,message) =>`

- This says create a new action, which takes two strings which we name username and message and use the delegate…

`GetMessage(username, message)));`

- Finally, we specify the method (that takes the two strings) that we will use in this instance to handle the GetMessage procedure call.

There are other (arguably more readable) ways of doing this but this helps break it down more clearly.

Let's hook up the connect button to make it connect to the server…

Return to the designer window and double-click on your connect button.

This will link a Click event to your button and scaffold you a method to handle the event in MainWindow.xaml.cs.

We will want this method to operate asynchronously (so it doesn't hang up our UI) so add the keyword `async` before the void return type in the method signature.

Inside the method add the code to start the connection:

```
try
{
    connection.StartAsync();
    MessagesListBox.Items.Add("Connection opened");
}
catch
{
    MessagesListBox.Items.Add("Connection failed");
}
```

Now the send button, to send a message…

Return to the designer window and double-click on your send button.

This will link a Click event to your button and scaffold you a method to handle the event in MainWindow.xaml.cs.

We will want this method to operate asynchronously (so it doesn't hang up our UI) so add the keyword `async` before the void return type in the method signature.

Into this method add the code to invoke a procedure on the server:

```
try
{
    await connection.InvokeAsync("BroadcastMessage", UsernameTextBox.Text,
MessageTextBox.Text);
}
catch (Exception ex)
{
    MessagesListBox.Items.Add(ex.Message);
}
```

Note: As before, ensure that the element names are correct as per your user interface (see yellow highlighting).

This code tries to invoke a procedure called "BroadcastMessage" on the hub, sending two strings.

As luck would have it, we've already created the method BroadcastMessage on the hub.

Now you have the basic functions of a SignalR client ready to go!

Our Server is set up and ready to go, our Client is set up and ready to go. Now it's a case of starting them and trying out our chat functionality.

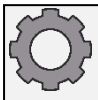
First let's run the client on its own.

- Right-click your client project and select **Debug -> Start New Instance**

Once loaded, click your connect button.

Those of you who are paying attention should realise that this should fail because our server is not running.

Notice how your UI shows "Connection opened" and no error… it is hiding the fact that something went wrong during the asynchronous call, which is bad. This is an illustration of why it is so important to pay attention when making asynchronous calls or engaging in multithreading.



Close your software, go back into your Client and insert the keyword 'await' in the line:

```
await connection.StartAsync();
```

- Run the Client on its own again and click your connect button again.

After a short timeout the underlying framework causes an exception to be thrown to our code, which we have handled and outputted a generic error message.

Reminder: await is a keyword which forces the execution of this code to pause until an asynchronous task has completed. It also returns control to its caller until the awaited task finishes. This is useful when order matters. In this case it would be inappropriate to call StartAsync() without an await because your code immediately moves on to add the item "Connection Opened" even before the StartAsync had finished. This could mean that the connection hadn't yet opened or might obscure the fact that an error will occur. However, you should be sure that your awaited task has its own timeout or add your own timeout otherwise, theoretically, you may be waiting for ever.

Now we're ready to test our functionality.

First, set up Visual Studio to start up both our Server and Client at the same time.

Make sure that the IIS Express profile is going to be used for the Server.

- Now Hit F5. Both your WPF app and your Web APP should start up.

Click Connect. You should see the message 'Connection opened'. If you don't, wait a few seconds for your web app to start up before trying again.

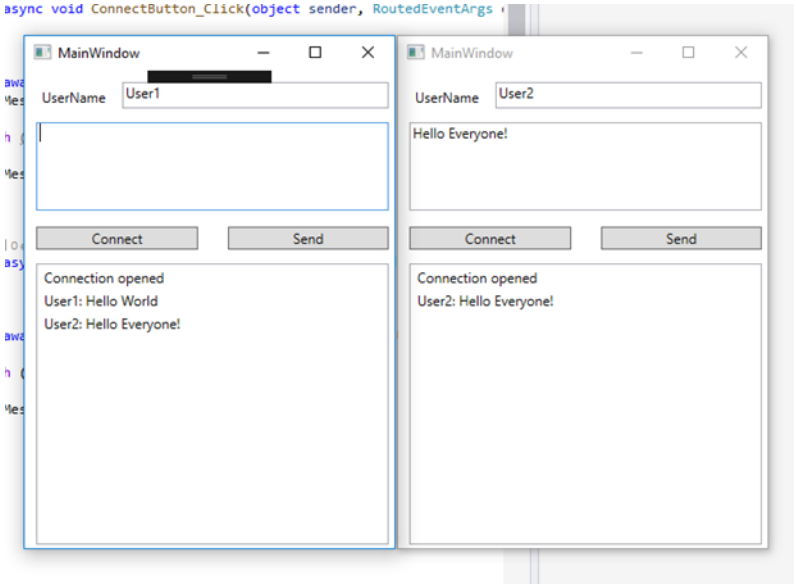
Now enter a Username and a Message and hit Send.

You should see your username and message appear in the ListBox.

Keep this client open but return to Visual Studio. Open the Solution Explorer window, right-click your client project and select Debug -> Start New Instance

You should now have two instances of the client running.

Click connect on the new client, enter a username and a message and press send. Your message is now sent to all clients.



You've now successfully created a basic SignalR service and client, which uses remote procedure calling to invoke functions at the endpoint.

Kestrel and Chatting with Others

So far we have been using IIS Express to host our server locally; but it would be more fun if we could run our chat app over the network.

In theory, this should be possible – but due to firewalls, security restrictions and other similar policy issues it's not certain to work in any specific environment.

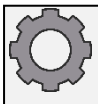
If you have multiple PCs, a friend with a PC connected to your LAN or you're on Campus let's give it a go anyway eh?

The first thing we should do is move from using IIS Express (a lightweight development version of Microsoft's Internet Information Services web server that is the backbone of most Microsoft Server hosts) to Kestrel (a small but powerful web server that is cross-platform).

Microsoft does not recommend that you expose Kestrel directly to the web because it doesn't have all of the functionality for filtering, managing, logging, limiting, scaling, etc. that you should really have in place. In production environments this is done by a server host environment. On Windows Server this might actually be Kestrel running within the full version of IIS, but theoretically Kestrel can run inside any web server environment.

For our testing and development purposes over LAN, Kestrel on its own will be just fine.

First let's move over to using Kestrel. The server is built into ASP.NET Core so when you deploy to a host Kestrel is already configured and ready to go as part of the project. Therefore, all we need to do is tell Visual Studio to run our project in Kestrel.



Click on your server project in the Solution Explorer window and, using the debug profile drop-down box, change the Profile from IIS Express to SignalRLab.

Now we need to know your IP address so Kestrel knows where to listen for connections. Open a Command Prompt window enter ipconfig and hit enter. You should look for a local IPV4 address (at home usually starts with 192.168… at Uni that starts with 150.237…)

Go back to launchSettings.json and, under the `SignalRLab` profile, replace the phrase 'localhost' with your IP address in the applicationUrl field.

Change the port to 5000

```
:hema: https://json.schemastore.org/launchsettings.json
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:48400",
      "sslPort": 0
    }
  },
  "profiles": {
    "SignalRLab": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://192.168.0.161:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Now let's make our chances of making this work a little higher by not requiring use of HTTPS – this means that we won't need to use an untrusted certificate. In a deployment environment we would, of course, require HTTPS but for development purposes on the LAN it isn't necessary. You may have already done some of this if you followed the steps above…

Open Program.cs and comment out the line: app.UseHttpsRedirection();

Finally, open your client and changethe WithUrl call to:

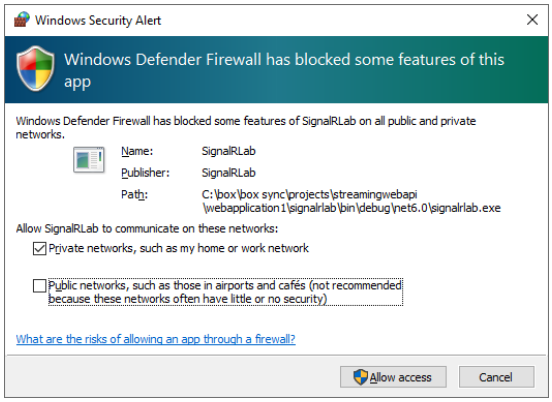
```
.WithUrl("http://192.168.0.161:5000/chatroom")
```

Make sure you replace the IP with the IP of your computer or the computer of someone else on the LAN who has configured their server for Kestrel.

Right-click the Solution in the Solution Explorer and click build.

Now test your Server / Clients with multiple machines. Try to get a small chat group between a few computers!

You may see the prompt below, in which case you will need to allow access to Private Networks.



Last Steps

1. Improve the functionality of your WPF app to lock in the username, clear the message box after sending, not activate the send box until a connection has been made, etc.

2. Add a mechanism by which the message is added to the local chat window of the sender before it has been sent to the server. Highlight this in orange to show that it hasn't yet been successfully broadcast. Once the client receives the broadcast, remove the highlight. If the client doesn't receive a broadcast with the message within a timeout period, highlight the message in red to show it was not successfully sent. Test this. (You may need to force your server to not respond to test this!)

3. Add some other functions to your server/client e.g. write methods in addition to BroadcastMessage which add a procedure to remove a message from all clients' message feed or a procedure which translates text once a message is sent.

4. What happens if the connection is accidentally closed because the client loses Internet? Research and add in functionality to handle this…

   You'll want to start by researching how to set a handler for the `connection.Closed` event