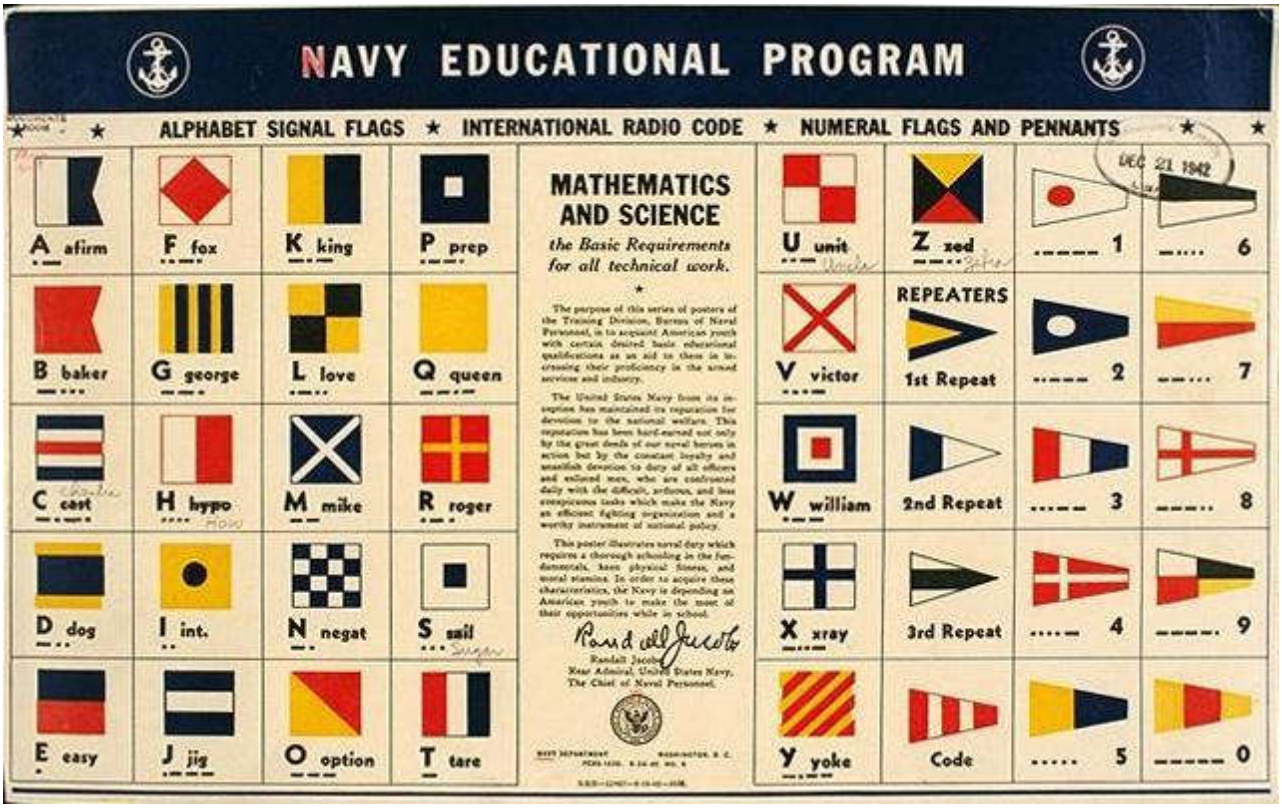
About SignalR Progress Demo



A simple ASP.NET Core2 app that uses SignalR to send progress messages from server-side code to an HTML page. Source code in github: SignalR-AspNetCore-ProgressDemo (https://github.com/endintiers/SignalR-AspNetCore-ProgressDemo) you can register feedback/comments there.

SignalR frees us from the request-response model so we can imagine other ways to interact between web pages, server-side code, services or other clients.

Code is truth, so lets have a look at that.

The Hub

All code in seperate classes under the SignalR folder in the solution. You can make ProgressInfo (the message definition) as complex as you like (as long as it's serializable). The Interface allows for injection later.

The actual Hub ReportProgress method is only intended to be called from the client (browser) - it can only send to it's caller (this.Context.ConnectionId).

```
public class ProgressInfo
{
    public string message { get; set; }
    public int pct { get; set; }
}

public interface IProgressHub
{
    Task ReportProgress(ProgressInfo info);
}

public class ProgressHub : Hub
{
    public Task ReportProgress(ProgressInfo info)
    {
        return Clients.Client(this.Context.ConnectionId).ReportProgress(info);
    }
}
```

At Startup

In Startup.cs. Wire up and start the ProgressHub.

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddSignalR();
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    ...
    app.UseSignalR(routes =>
    {
        routes.MapHub<ProgressHub>("prog");
    });
}
```

In the page

Unobtrusive Ajax is used to aysnchronously POST the form to the server while still taking advantage of Razor Page routing and binding. Let's just call this 'magic' and move on :-).

```
(in _Layout.cshtml)
<script src="~/js/jquery.unobtrusive-ajax.min.js"></script>
...
(in Index.cshtml)
<form asp-page-handler="OnPost" data-ajax="true" data-ajax-method="POST">
```

The **connection** variable is global - this reference is used later to retrieve the connection id.

On document ready (called only once - after the initial GET) a connection to the SignalR hub is created, and a function is defined to handle reportprogress messages. The connection is then started and the page displayed.

```
(in _Layout.cshtml)
<script src="~/js/signalr-client-1.0.0-alpha2-final.min.js"></script>
...
(scripts section of Index.cshtml)
<script>
        var connection;

        $('document').ready(function () {
            connection = new signalR.HubConnection('/prog');
            connection.on('reportprogress', info => {
                console.log(info.message + ' - ' + info.pct + '%');
                reportProgress(info);
            });
            connection.start();
        });
```

Because the form POST is asynchronous we need a way to reset the state of the page once processing is finished. When the server sends a progress report with the message 'Reset' the UI state is re-initialised. Other messages just set the submit button state to loading (over and over...meh) and update the progress bar.

```
        function reportProgress(info) {
            if (info.pct < 1 || info.message.toLowerCase() == 'reset') {
                $('#pbar').css('width', '0%')
                    .attr('aria-valuenow', 0).text('');
                $('#progressButton').button('reset');
            }
            else {
                $('#pbar').css('width', info.pct + '%')
                    .attr('aria-valuenow', info.pct).text(info.message);
                $('#progressButton').button('loading');
            }
        }
```

On form submit we need to ensure that the model bound hidden input variable in the form ('connectionId') is populated. We need the connection id so the server knows who to inform about the progress of the long-running operation.

The hub connection (connection) has an Id property, but this always seems to be 0 in this version of the SignalR JavaScript client. The Id value can also be retrieved from connection.connection.connectionId (hubConnection.httpConnection.connectionId).

Just to show that we can also send messages from a JavaScript client ReportProgress is called from here to set '10% complete'. This is the only time that the Hub's actual ReportProgress method is executed. The message goes through a socket from the page to the hub (on the server) and then back to the same page.

```
        function getConnectionIdAndReportStart() {
            $('#connectionId').val(connection.connection.connectionId);
            var info = { message: 'Starting Out', pct: 10 };
            connection.invoke('reportprogress', info);
        }
</script>
```

In the Page code-behind

We are allowed to call it code-behind aren't we?

The Page Model's constructor takes an IHubContext. AspNetCore2 injects a reference to the ProgressHub instance for us. Now we can make calls to the hub from here! On POST the ConnectionId of the caller is bound from a hidden input on the page, so we are all ready to go.

In the server-side code we can now do lots of different work while reporting to the client what we are doing and how it's going via SignalR. Here I am just updating a ProgressBar, in my actual app (**Unearth**) I do all kinds of things like parsing and analysing natural language, executing several searches on different repositories and passing the results back to the client as I go.

This is all done with *_progressHubContext.Clients.Client(connectionId).ReportProgress(info);* This uses the hub, but doesn't actually execute the Hub's ReportProgress method.

When our long-running method is finished a 'Reset' message is sent to the client which will cause it to reset it's state ready for the next call.

```
public class IndexModel : PageModel
{
    IHubContext<ProgressHub, IProgressHub> _progressHubContext;
    public IndexModel(IHubContext<ProgressHub, IProgressHub> progressHubContext)
    {
        _progressHubContext = progressHubContext;
    }

    [BindProperty]
    public string ConnectionId { get; set; }

    ...

    public void OnPost()
    {
        // 10% report is done in js code...
        Thread.Sleep(1000);
        ReportAndSleep("Relaxing Splines", 20, ConnectionId, 1000);
        ...
        ReportAndSleep("Reset", 0, ConnectionId, 0);
    }

    private void ReportAndSleep(string message, int pct, string connectionId, int sleepF
or)
    {
        var info = new ProgressInfo() { message = message, pct = pct };
        _progressHubContext.Clients.Client(connectionId).ReportProgress(info);
        Thread.Sleep(sleepFor);
    }
}
```

You might wonder why the server-side method signature is `public void OnPost()` rather than `public async Task OnPostAsync()` . It could be either. The client-side call is asynchronous but the server-side code can be synchronous if you like. The page isn't waiting for us (but the Progress/Submit button there is disabled after the first message is received).

For a progress bar example synchronous code on the server makes sense - but the ability for the server to send messages back to the page allows us to imagine much more complex scenarios. If the server-side method is asynchronous we can launch any number of independent operations, passing the hub reference and client connection id and updating our UI from within those operations when they are complete. I'd like to re-write the **Webjet** code now :-).