CREATING an hello-world MFE

Step 1- create a regular React application

```
npx create-react-app micro-app-1 --template typescript
```

When this has been created, open it in VS code.

Step 2 – create the component to be made available as a remote micro-app

Set up a folder under src called HelloWorld and a file called HelloWorld.tsx

HelloWorld.tsx

```
const HelloWorld = () : JSX.Element => {

    return <p>Hello from micro app 1!</p>
}

export default HelloWorld;
```

Step 3 – use the component and run it as a normal React application

Edit App.tsx

App.tsx

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import HelloWorld from './HelloWorld/HelloWorld';

function App() {
  return (
    <div>
      <h1>This is the micro-app running</h1>
      <HelloWorld />
    </div>
  );
```

```
}


export default App;
```

Use npm start to test this is working, then stop this running.

At this point we have just done React, nothing special!

Step 4 – convert our project to build with webpack

We do get webpack as part of React but we need a few extra packages to make the module federation plugin parts work.

Add these to the dependencies section of package.json:

```
    "webpack-cli": "^5.1.0",
    "webpack-merge": "^5.10.0",
    "html-webpack-plugin": "^5.5.0",
    "ts-loader": "^9.1.2"
```

And then run npm install

At this point we are going to change the way that our application works. Instead of starting our application using the regular react commands we are instead going to switch to webpack.

The first step is to provide some configuration…

We'll create a file called webpack.config.js and place it in the root of our application. Note that this is the standard file name and location that webpack will look for – if you want to call it something different or place it in a subfolder you can but you'll need to adjust the startup script to say where to find it (we'll come back to that in a few minutes).

The config file looks like this:

webpack.config.js - NOTE THIS FILE IS AVIALBLE TO COPY AND EDIT FROM THE RESOURCES FOLDER!

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
const ModuleFederationPlugin =
require('webpack/lib/container/ModuleFederationPlugin');

module.exports = {
    mode : 'development',
```

```
    devServer : {
        port : 3001,
        historyApiFallback : {
            index : '/index.html'
        },
    },
    resolve: {
        extensions: [".ts", ".tsx", ".js", ".jsx"],
      },
plugins : [
        new HtmlWebpackPlugin({
            template : './public/index.html'
        }),
        new ModuleFederationPlugin({
            name : 'helloworld',
            filename : 'remoteEntry.js',
            exposes :
                {'./HelloWorld' : './src/index'}
        }
        )
    ],
    module: {
        rules: [
            {
                test: /\.(js|jsx|tsx|ts)$/,
                loader: "ts-loader",
                exclude: /node_modules/,
            },
            {
                test: /\.css$/,
                exclude: /node_modules/,
                use: [ 'style-loader', 'css-loader' ]
            }
        ]
    }
};
```

This will be in future a copy and paste job but we'll talk through what each of the important parts of this file means…

```
    mode : 'development',
```

By default we will run in development mode when we do npm start. This will spin up a web server with some specified settings which will be useful. This will not apply if we choose to run in production mode, and we'll mention how to do that in the next step.

```
devServer : {
        port : 3001,
        historyApiFallback : {
            index : '/index.html'
        },
    },
```

While running our development server, it'll be on port 3001 (or whatever we choose) – we need a unique port for each micro-app. This won't apply when we deploy to a live server – it'll be managed by our web server proxy settings.

If a user tries to visit a URL other than the root of the application, they'll get index.html. We want that to happen because that's the requirement for React routing which we will implement later – if we go to localhost/somepage in our browser we want index.html to be served up every time.

```
    resolve: {
        extensions: [".ts", ".tsx", ".js", ".jsx"],
      },
```

This is telling webpack to look at all code files with these extensions. It's only needed because we're using typescript… by default webpack would look at js and jsx!

```
        new ModuleFederationPlugin({
            name : 'helloworld',
            filename : 'remoteEntry.js',
            exposes :
                {'./HelloWorld' : './src/index'}
        }
```

This is the important bit – it's saying that our HOST will be able to access afile called remoteEntry.js (or rather the url localhost:3001/remoteEntry.js) – we won't create that file, webpack will do it for us.

The file will contain a namespace of helloworld (that's the name part) and a component called HelloWorld… it can find the component by going to the src/index.tsx file. Actually we are going to change this to improve it a bit, but the principle here is that this is where we define what is available as a remote component.

The rest of this file Is a standard copy and paste which tells webpack how to process different file types. We'll not edit this.

Having created a configuration we can do the final step in this part which is to change our scripts to use webpack instead of react. In packackge.json:

```
  "scripts": {
```

```
    "start": "webpack serve",
    "build": " webpack --mode production",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
```

Step 5 – test and fix some issues

We've not quite finished but this is a good stage to try running this and seeing if we have any issues to fix (we will have!)

Do npm start

In our console it will say:

```
ERROR in ./src/index.tsx
Module build failed (from ./node_modules/ts-loader/index.js):
Error: TypeScript emitted no output for D:\gitrepositories\neu
    at makeSourceMapAndFinish (D:\gitrepositories\neueda\react
```

The message here is "Typescript emitted no output". This is actually not a problem – the issue is that our default react application has some lint settings that expect output when there might not always be some, so to remove this error we need to edit tsconfig.json

```
    "noEmit": false,
```

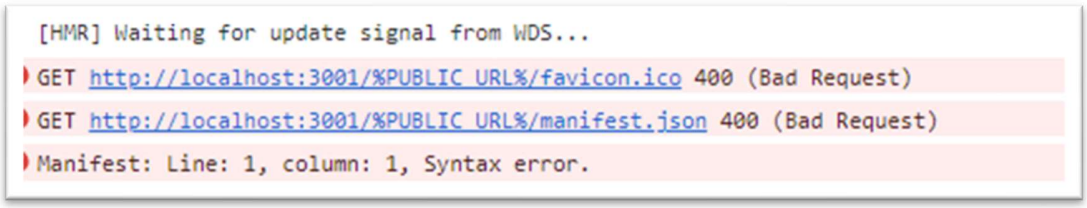
Stop & restart npm start

At this point there should be no errors in the VScode console.

Visit http://localhost:3001/ in the browser.

The application should work but you'll have an error in the browser console & the VSCode console saying that it can't resolve %PUBLIC_URL%

```
JRIError: Failed to decode param '/%PUBLIC_URL%/favicon.ico'
    at decodeURIComponent (<anonymous>)
    at decode_param (D:\gitrepositories\neueda\react-mfes\hell
    at Layer.match (D:\gitrepositories\neueda\react-mfes\hello
```



In public/index.html we have a reference 3 times to a variable called PUBLIC_URL – this is set by react but not by webpack. We won't try to fix that in any kind of clever way, we'll just remove the references to it. We can use this as an opportunity to tidy up this file too.

public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="/logo192.png" />
    <link rel="manifest" href="/manifest.json" />
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

We now should have the app running with no errors.

Step 6 – expose the remote component

At this stage we are running the HelloWorld component in a local container – that's perfect for development but we've got a few further steps to make this available remotely – in theory the entire application would be available remotely – we are exposing index.tsx. we don't want to do that – we want

this and app.tsx to be a local development container, and only our HelloWorld component to be exposed remotely.

There are 2 things we need to do:

First, we'll create a file that replaces index.tsx for use remotely. This won't be quite the same as index.tsx – in this version we look for our root div and load the components there. In the one that we'll expose we'll create a function that our container can call. The function will be called mount and it will receive from the container a div element as a property. Then it can render our HelloWorld component in that div.

src/helloWorld/indexHelloWorld.tsx

```
import ReactDOM from 'react-dom/client';
import './../index.css';
import HelloWorld from './HelloWorld';

const mount = (el: Element) : void => {

  const root : ReactDOM.Root = ReactDOM.createRoot(el);

  root.render(<HelloWorld />);
}

export { mount };
```

Now we can go back to our webpack config file and update the exposes line to use this file instead of index.tsx

```
            exposes :
                {'./HelloWorld' : './src/HelloWorld/indexHelloWorld'}
```

Step 7 – check everything is working…

If we now do

npm start

we should be able to visit

http://localhost:3001/

and also check that this url resolves – that's the file that our container will access.

http://localhost:3001/remoteEntry.js

We are now done. There will be a tweak needed to make it work still, but we'll find out about these when we work on the container application so that we can see the error message first.

Just 2 further things before we leave this:

(1) Point out that in package.json when you do a build you override the mode to production. You can do npm run build and check what is put into the dist folder
(2) If we want to provide an alternative config file for webpack (either a different name or in a different folder, you can specify this as part of the script too, eg:

```
"start": "webpack serve --config config/webpack.dev.js",
```

CREATING THE CONTAINER

Step 1- create a regular React application

```
npx create-react-app container --template typescript
```

When this has been created, open it in VS code.

Step 2- add the webpack dependencies

Add the same dependencies into package.json as we did for the remote micro-app

```
    "webpack-cli": "^5.1.0",
    "webpack-merge": "^5.10.0",
    "html-webpack-plugin": "^5.5.0",
    "ts-loader": "^9.1.2"
```

Then run npm install

Step 3 – create webpack.config.js

Again there is an outline of this file in the resources section.

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
const ModuleFederationPlugin =
require('webpack/lib/container/ModuleFederationPlugin');

module.exports = {
    mode : 'development',
```

```
    devServer : {
        port : 3000,
        historyApiFallback : {
            index : '/index.html'
        },
    },
    resolve: {
        extensions: [".ts", ".tsx", ".js", ".jsx"],
      },
    plugins : [
        new HtmlWebpackPlugin({
            template : './public/index.html'
        }),
        new ModuleFederationPlugin( {
            name : 'container',
            remotes : {
                helloworld : 'helloworld@http://localhost:3001/remoteEntry.js'
            },
            shared : ['react', 'react-dom']
        })
    ],
    module: {
        rules: [
            {
                test: /\.(js|jsx|tsx|ts)$/,
                loader: "ts-loader",
                exclude: /node_modules/,
            },
            {
                test: /\.css$/,
                exclude: /node_modules/,
                use: [ 'style-loader', 'css-loader' ]
            }
        ]
    }
};
```

Point out the ModuleFederationPlugin – this time we list our remotes and where to find them.

```
helloworld : 'helloworld@http://localhost:3001/remoteEntry.js'
```

The “helloworld” that’s before “@http…” needs to match the name in the webpack config of the client.

The “helloworld” at the start of the line is how we will reference it internally – these will normally match but in case you had two separate external sources with the same name, we could use a different name here.

Step 4 – edit the startup / build scripts

The next step is to edit package.json so that we use webpack rather than react:

```
  "scripts": {
    "start": "webpack serve",
    "build": " webpack --mode production",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
```

Step 5 – attempt to use the remote module

We are now ready to try and use the remote module. In App.tsx let’s try and import it.

```
import {mount} from 'helloworld/HelloWorld';
```

This line won’t compile – typescript lint is complaining that it can’t find the module. We need to tell our application that it exists – webpack will substitute the real one at runtime, so we’ll just tell it that this is a module that exists.

To do this create a new file in src called remotecomponents.d.ts

The .d.ts part takes this a type declaration file – these files are only used to enable the compile time type checking – we normally use them to allow us to use javascript modules in typescript.. and that’s what we have here – although we wrote them in typescript they’re transpiled to JS!

remote-modules.d.ts

```
export declare module 'helloworld/HelloWorld';
```

Now the import in our App.tsx file should compile.

SJ – change noEmit to false in tsconfig.json file, remove logo related bits from app.jsx

To actually use it, we need to execute the mount function. The mount function expects a parameter which is an element it can render into. We therefore need to get access to an element. We do this with useRef – note that we need to do the mount in a useEffect block so that it runs after the ref has been set up.

App.tsx

```
import React, { useEffect, useRef } from 'react';
import logo from './logo.svg';
import './App.css';
import {mount} from 'helloworld/HelloWorld';

function App() {

  const helloWorldDiv = useRef<HTMLDivElement>(null);

  useEffect( () => {
    mount(helloWorldDiv.current);
  } , []);

  return (
    <div>
      <h1>This is the container</h1>
      <div ref={helloWorldDiv} />
    </div>
  );
}

export default App;
```

Step 6 – check everything is working…

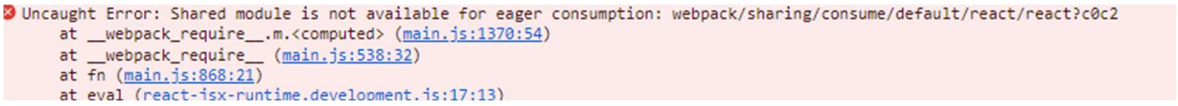
If we now run npm start, we'll see the same error we saw earlier saying there was no output emitted from typescript – the same fix is needed – change the following line in tsconfig.json:

```
    "noEmit": false,
```

Then re start the application.

This should now start with no issues in the console – visit localhost:3000

In the browser's console we will see an error message that says that the shared module is not available for eager consumption



This error occurs because "things are not quite ready" – the actual issue is a bit more complicated to do with synchronous and asynchronous but the fix is quite easy. What we need to do is have a file load before index.tsx's code runs… the fix is:

First rename index.tsx to bootstrap.tsx

Then create a new index.tsx file with the following contents:

```
import ('./bootstrap);
```

Be careful to include the brackets and don't include the .tsx extension or this won't work. What this actually does is as follows: when our application runs it runs index.tsx. The line of code in index.tsx is running a function called import - that's not the same as just importing a file. What this function does it inspects the file it is importing and loads any dependencies that file has, then it runs the file. That's why the brackets are needed, and it's why now things start to work.

We also need to make another change in the tsconfig.json file:

```
    "isolatedModules": false,
```

Now if we restart and refresh the browser we'll see the error about not being able to parse %PUBLIC_URL% so again we'll edit the public/index.html file to remove the references to these

public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="/logo192.png" />
    <link rel="manifest" href="/manifest.json" />
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

Finally we'll have the warning about createRoot being called twice – in bootstrap.tsx remove the strict mode.

bootstrap.tsx

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
    <App />
);
```

We now have our first working container which is using our first working micro-app.

To prove it's working, stop the micro-app from running and refresh the browser.

Step 7 – Making things more standard

When we want to import multiple remote modules, because we'll always call the function that they export mount, it will be difficult to import multiple modules into a single component in React. So it makes things easier if we create wrapper modules…

In the container application src folder

- Create a new folder called remoteComponents
- Create a file in this folder called HelloWorld.tsx

Make this a component and then move the relevant code into this file:

HelloWorld.tsx

```
import React, { useEffect, useRef } from 'react';
import {mount} from 'helloworld/HelloWorld';

const HelloWorld = () : JSX.Element => {

  const helloWorldDiv = useRef<HTMLDivElement>(null);

  useEffect( () => {
    mount(helloWorldDiv.current);
  } , []);

  return (
    <div ref={helloWorldDiv} />
  );
```

```
}

export default HelloWorld;
```

Now we can use this component like any other, so App.tsx becomes:

```
import './App.css';
import HelloWorld from './remoteComponents/HelloWorld';

function App() {
  return (
    <div>
      <h1>This is the container</h1>
      <HelloWorld />
    </div>
  );
}

export default App;
```