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,
```

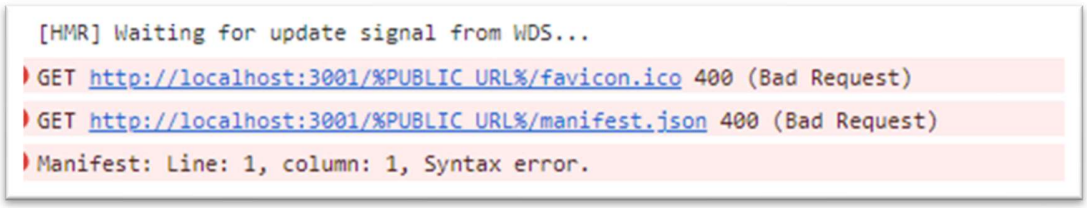
Stop & restart npm start

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.