Part 2 – Setting up Webpack to create a JavaScript Web resource bundle

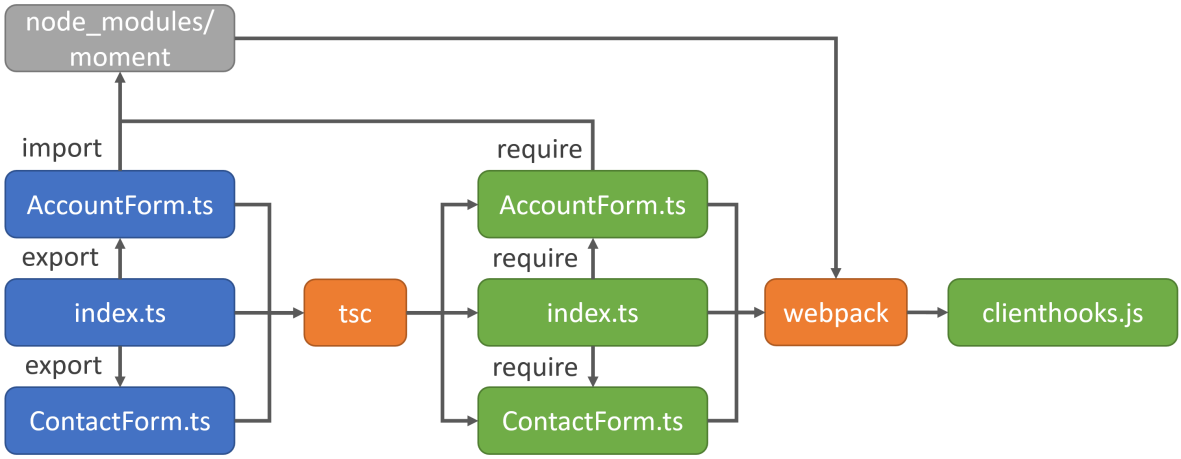
This is part of the course 'Scott's guide to building Power Apps JavaScript Web Resources using TypeScript'.

In this second part, we will cover how to create your Web Resource JavaScript from your TypeScript project.

When writing TypeScript, usually the following is true:

1. You span your TypeScript across multiple source files that need to be compiled into a single JavaScript file
2. You import modules from `node_modules` that need to be also compiled into the single JavaScript file
3. Your JavaScript typically will include source maps to enable debugging during development, but should be minified (optimised in size) for production distribution.

Webpack is a tool that addresses each of these issues by taking the output from the TypeScript compiler (`tsc`), parses the files and bundles them along with any required node modules into a single JavaScript file that can be run inside the browser.



Babel vs TypeScript JavaScript Transpilation

`Webpack` can be used in conjunction with another library called `Babel`. In a similar way to the TypeScript compiler (`tsc`), Babel can take your TypeScript code and translate it into JavaScript (often referred to as transpiling) so that it will run inside the browser. You might use Babel if you needed more control over the final output such as adding polyfills, using features that the TypeScript compiler doesn't support or provide multiple versions targeting different run-times. I find no need to use Babel for creating JavaScript Webresources. If you are using Babel then make sure you are using a loader that runs the TypeScript compiler to perform type checking since Babel on it's own will simple convert to JavaScript without performing any checks.

Using the `ts-loader` webpack plugin will run the tsc compiler for you so that all your types are checked – but if you were using an alternative Babel loader, you can always run `tsc --noEmit` to run the TypeScript compiler and show errors, but not generate any output.

Install & Configure Webpack

Run the following at the command line of your VSCode Project:

```
npm install webpack webpack-cli webpack-merge ts-loader --save-dev
```

Download and copy the following Webpack configuration files to your project folder:

webpack.common.js

webpack.dev.js

webpack.prod.js

The core webpack configuration is found in `webpack.common.js` This file is automatically used by webpack. For Dev and Prod build, the common configuration is merged with the specific settings (using `webpack-merge`) for the target environment.

Open the `webpack.common.js` and edit the library settings, to define the filename and your JavaScript webresource namespace.

```
filename: "bundle-name.js",
library: ["namespacepart1","namespacepart2"]
```

In our project this looks like this:

```
filename: "ClientHooks.js",
// Set this to your namespace e.g. cds.ClientHooks
library: ["cds", "ClientHooks"],
```

If you simply wanted to namespace your code under foo, you could use:

```
library: ["foo"],
```

This would expose your code under the library name `foo`

Configuring webpack to run using `package.json` scripts

Open the `package.json` file and add the following scripts, overwriting the existing scripts section created by `npm --init`

```
"scripts": {
    "build": "webpack --config webpack.dev.js",
    "start": "webpack --config webpack.dev.js --watch",
    "dist": "webpack --config webpack.prod.js"
},
```

This allows us to run the development build at the command line using:

```
npm start
```

If wanted to create a distribution production build, we would use:

```
npm run-script dist
```

Later in this series, we will create unit-test and integration-test scripts and add them to this section of the `package.json`.

Create an entry point.

A key feature of webpack is that it will determines which modules are required to be bundled into the output JavaScript by looking at your code and the modules that it imports. To determine what is needed to be bundled, an entry point must be provided. The default is `index.ts`.

> **Note:** Since TypeScript 1.8, the tsc compiler has had the `-outFile` that creates a single bundled file similar to Webpack, however webpack gives much finer control over how this bundle is generated.

Create a file in the src folder named `index.ts` and add the following TypeScript:

```
export * from "./Forms/AccountForm";
```

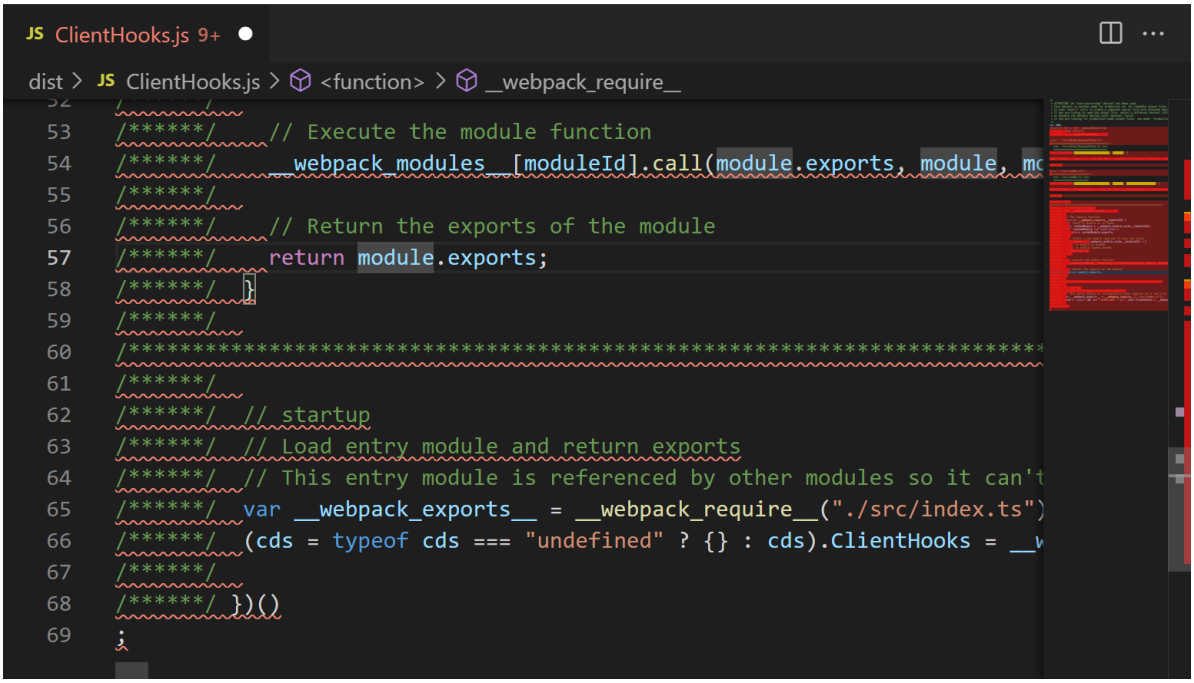
Add the command line, type:

```
npm start
```

This will start webpack in watch mode, so that as you change your code the bundle will incrementally be compiled. You will find the output JavaScript in `dist/ClientHooks.js`

> **Note:** You can change the name of entry point and the output file in the `webpack.common.js` file.

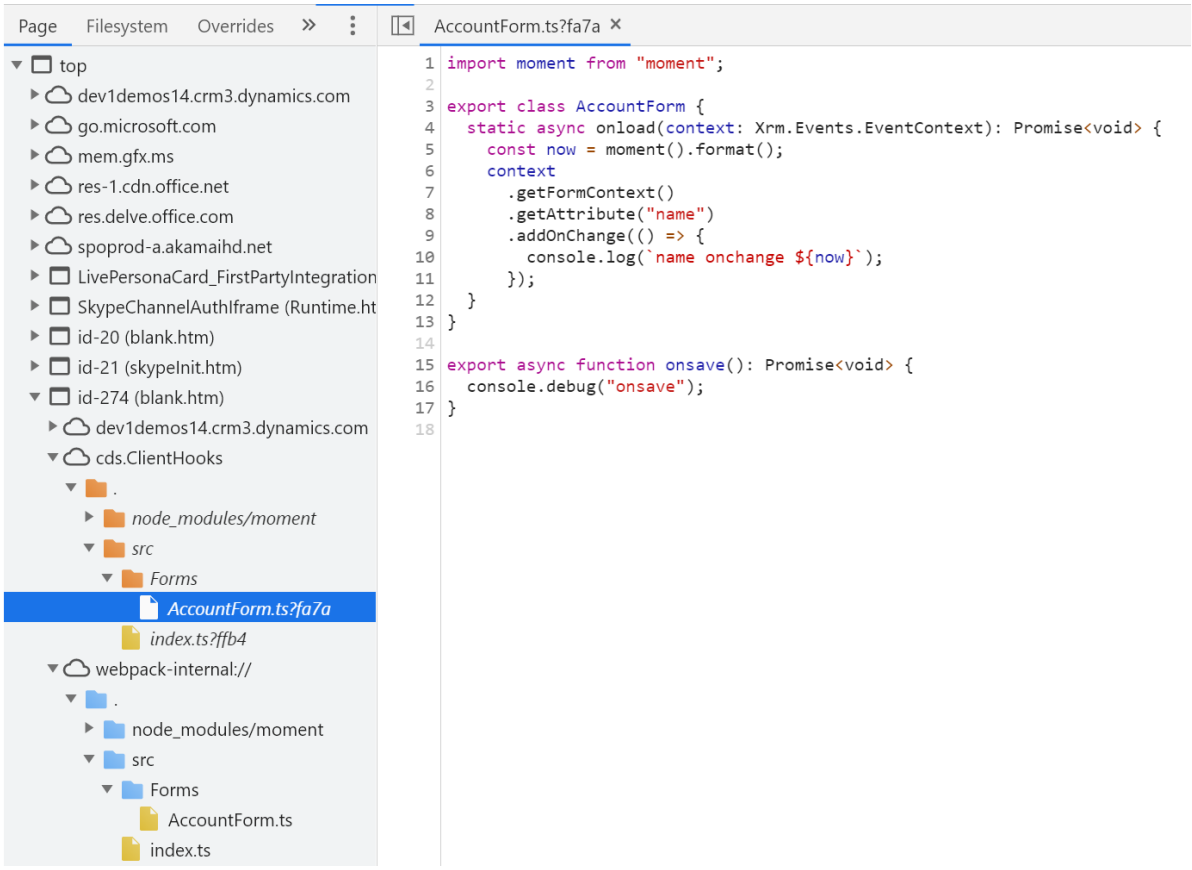
If you look at the `ClientHooks.js` you will see that it looks very different to your TypeScript files. This is because it is transpiled to `ES5` JavaScript and includes source maps. Source maps provide the information that is needed to debug the TypeScript code even though it is actually JavaScript being executed. If you are used to debugging c# you can think of the source-maps in a similar way to the `.pdb` files created in Debug configuration builds of c# code.



> **Note:** The Underlined red code is because this code does not conform to ESLint's formatting rules.

If you search for `sourceMappingURL=` you will find your original TypeScript code base64 encoded. Later we will see how you can step through your TypeScript code in the browser via the webpack: source files even though it is actually JavaScript running.

> Note: The original TypeScript output is also visible as source files under the **webpack-internal:** prefix – but normally you do not need to view these source maps – it will be the source files prefixed with **webpack:** that we are interested in.



If you make a change to the `AccountForm.ts` you will see that it is detected, and the `dist/ClientHooks.js` file is updated:

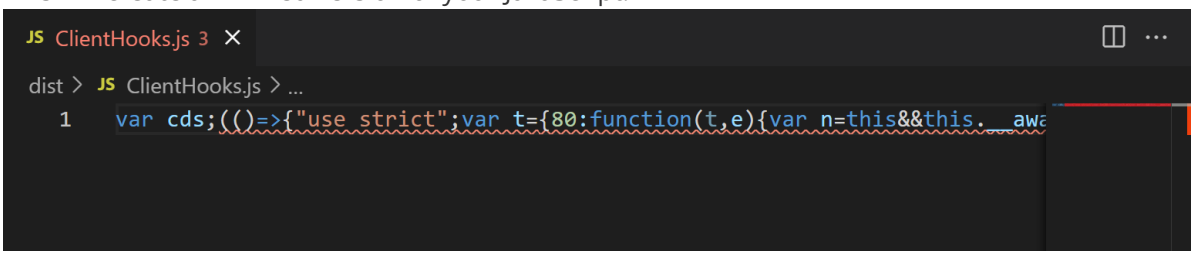
```
./src/index.ts 657 bytes [built] [code generated]
./src/Forms/AccountForm.ts 3.16 KiB [built] [code generated]
webpack 5.36.2 compiled successfully in 1724 ms
asset ClientHooks.js 13.7 KiB [emitted] (name: main)
./src/index.ts 657 bytes [built] [code generated]
./src/Forms/AccountForm.ts 3.16 KiB [built] [code generated]
webpack 5.36.2 compiled successfully in 140 ms
```

If you added additional modules that must be exported that are not themselves referenced by the `AccountForm.ts`, then they must be included in the `index.ts`. If however you simply import them inside the `AccountForm.ts`, they will be automatically detected and bundled. We will see this in action later in this series.

If you press `Ctrl + C` to stop the watch process, you can then create a production build by typing

```
npm run-script dist
```

This will create a minified version of your JavaScript:



Later we will see how to deploy these JavaScript files into a Model Driven App.

We can see the Module bundling take place by adding the `moment` library so that we can manipulate and parse dates. At the command line, type:

```
npm install moment
```

You will see in the `packages.json` a new entry that defines the required module. This will be installed later if needed during a build by using `npm install`.

package.json > ...

```
{
  "name": "clientjs",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷ Debug
  "scripts": {
    "build": "webpack --config webpack.dev.js",
    "start": "webpack --config webpack.dev.js --watch",
    "dist": "webpack --config webpack.prod.js"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/xrm": "^9.0.39",
    "@typescript-eslint/eslint-plugin": "^4.22.1",
    "@typescript-eslint/parser": "^4.22.1",
    "eslint": "^7.25.0",
    "eslint-config-prettier": "^8.3.0",
    "eslint-plugin-prettier": "^3.4.0",
    "eslint-plugin-react": "^7.23.2",
    "prettier": "2.2.1",
    "ts-loader": "^9.1.2",
    "typescript": "^4.2.4",
    "webpack": "^5.36.2",
    "webpack-cli": "^4.7.0",
    "webpack-merge": "^5.7.3"
  },
  "dependencies": {
    "moment": "^2.29.1"
  }
}
```

You can now use this module in the `AccountForm.ts`:

```
import moment from "moment";
export class AccountForm {
  static async onload(context: Xrm.Events.EventContext): Promise<void> {
    const now = moment().format();
    context
      .getFormContext()
      .getAttribute("name")
      .addOnChange(() => {
        console.log(`name onchange ${now}`);
      });
  }
}
```

If you run `npm start`, you will now see the `dist/ClientHooks.js` has the moment library bundled into it. A key principle to bundling is that internal modules are not exposed in the global namespace. This allows for multiple versions to be loaded side-by-side at runtime if needed by different components.

> **Note:** It used to be a common practice to load additional modules on demand rather than bundling - this practice is generally accepted as not best practice due to the additional http requests required and deployment complexity.

ES5 vs ES6+

TypeScript allows us to write code using some of the rich language features of the more recent JavaScript standards. When the `tsc` compiler is used to combine into JavaScript, you can target which version of JavaScript you want to support using the target configuration of the `tsconfig.json` file. Currently if you need to support IE11 (which is still used by some version of the outlook App), then the `tscofing.json` must specify targeting `es5`.

```
{
  "compilerOptions": {
    "target": "es5",
    ...
```

One example of this is the use of the `async` / `await` pattern. This allows us to easily call asynchronous code that returns a `Promise` without the need for call-backs. If the `tsconfig.ts` defines the output to be es5, since `await` is not part of the es5 standard it is compiled into JavaScript using the older call-back style code.

As browsers support more modern features, the target can be updated to output JavaScript according to a later standard, and consequently will be much closer to the original TypeScript. See the following table for the features supported by browsers that are part of es6 https://kangax.github.io/compat-table/es6/

You can change the target to es6 if you do not need to support IE11. This will reduce the complexity of the transpiled JavaScript since it can use language features such as classes.

```
{
  "compilerOptions": {
    "target": "es6",
    ...
```

Tree shaking

One advantage of using `webpack` is something called tree shaking . If you are importing a module that has many different exports but you only use some of them, Webpack will work out that it only needs to bundle the code that you use. This can reduce the size of you bundled JavaScript, but the extent to which it makes a difference depends on the module structure of the modules you are consuming. See https://bluepnume.medium.com/javascript-tree-shaking-like-a-pro-7bf96e139eb7

To ensure that tree shaking can be used on your code, the `tsconfig.json` must include:

```
"module": "es2015"
```

If you use `commonjs` (the default), then any import/exports will be converted to code that webpack will not be use to tree shake.
IMPORTANT: You will only see tree shaking happen when you do a production build of your code - the development build will not remove un-used exports.

> **Note:** Tree shaking will only have a significant effect on your bundle size if you are using very large libraries that have many components that you are not using, or if your TypeScript code contains large amounts of redundant exports that are not used.

Up Next...

In the next part we will look at adding unit-tests using a library called `jest`.