ES6 modules, now in a browser near you

There are some news and changes since I last visited ES6 modules. I started hearing more and more about WebPack and how it did the one thing that Rollup doesn't... it splits the code in different bundles based on the overall size of your Javasacrip files.

The latest news is that modules are now supported in browsers so packaging them may or may not be as necessary any more.

ES6 modules defined

In ES6 each module is defined in its own file. The functions or variables defined in a module are not visible outside unless you explicitly export them. This means that you can write code in your module and only export those values which should be accessed by other parts of your app.

ES6 modules are declarative in nature. To export certain variables from a module you just use the keyword export. Similarly, to consume the exported variables in a different module you use import.

Rollup

Things haven't changed since the last time I worked with Rollup so I'll use the same example I used then.

The first part of the example is the notional image manipulations. We load the image using a promise and resolve it on load and reject with a new error. The other functions just log what they would to to console.

Using the `export` keyword tells the parsers that the function can be imported from other scripts as we'll do later in our main script.

```
// image-manip.js
export function loadImage(url) {
```

```
  return new Promise( (resolve, reject) =>{
    var image = new Image();
    image.src = url;

    image.onload = () => {
      resolve(image);
    };

    image.onerror = () => {
      reject(new Error('Could not load image at ' + url));
    };
  });
}

export function scaleToFit(width, height, image) {
    console.log('Scaling image to ' + width + ' x ' + height);
    return image;
}

export function watermark(text, image) {
    console.log('Watermarking image with ' + text);
    return image;
}

export function grayscale(image) {
    console.log('Converting image to grayscale');
    return image;
}
```

Our main script uses the `import` keyword to bring the exported fnctions from the image-manip library into the current file. We then use it to create a function without having to redefine the functions, we just use them.

```
// app.js
import { loadImage, scaleToFit, watermark, grayscale } from './image-'./i
function processImage(image) {
  loadImage(image)
```

```
    .then((image)  => {
      document.body.appendChild(image);
      return scaleToFit(300, 450, image);
    })
    .then((image)  => {
      return watermark('The Real Estate Company', image);
    })
    .then((image)  => {
      return grayscale(image);
    })
    .catch((error) => {
      console.log('we had a problem in running processImage ', error);
    });
}
```

To create a Rollup bundle we need to do the following:

1. Install Rollup globally
2. Install Rollup and related plugins at the project level
3. Create a `rollup.config.js` configuration file
4. Run the command line, configure your package file to run the build through NPM or your build system

NPM installation is pretty straight forward. This will take care of steps 1 and 2 of our task list.

```
npm install -g rollup

npm install -D rollup \
babel-preset-es2015-rollup\
rollup-plugin-babel \
rollup-plugin-commonjs \
rollup-plugin-json \
rollup-plugin-node-resolve
```

The configuration file tells Rollup what files to work with and how to process them. This is step 3 in our task list.

```
'use strict';

import commonjs from 'rollup-plug'rollup-plugin-commonjs'eResolve from 'ro
import json from 'rollup-plugin-json';

ex'rollup-plugin-node-resolve's6/main.js',
  plugins: [
    nodeResolve({
      jsnext: true
    }),
    commonjs({
      include: 'node_modules/**'
    }),
    json()
  ],
  dest: './bundle.js'
};
'node_modules/**'
```

How we run the command depends on the tooling we have set up. We can run Rollup from the command line using a command like the one below. This command assumes that there is a `rollup.config.js` in the directory where we run the command.

```
rollup -c --output bundle-2.js # --output is equivalent to dest
```

We can also configure NPM to run our Rollup build as a script. In the example below we run the command using `npm run build` which will then trigger the build sing the configuration available in the same directory.

```
{
  ...,
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "rollup -c"
  },
```

```
  ...
}
```

The final version is a Gulp task to handle transpilation to ES5 and run Rollup to create the bundle. This is pretty close to the configuration file but split differently to accomodate the two step process. Soon creating bundles for browsers will become easier because we won't need the transpilation process... browers are beginning to support modules natively.

```
const gulp = require('gulp');
const rollup = require('rollup')
;

gulp.task('bui'rollup'ction () {
  return rollup.rollup({
    entry: "./src/main.js",
      plugins: [
        nodeResolve({
          jsnext: true
         "./src/main.js"commonjs({
          include: 'node_modules/**'
        }),
        json()
      ],
  })
    .then(function (bundle) {
      bundle.write({
        format: "umd",
        moduleName: "library",
        dest: "./bundle.js",
        sourceMap: true
      });
    })
});
'node_modules/**'
```

Webpack

Webpack is the 500 pound gorilla when it comes to compressing and bundling web content. It works on both CSS and Javascript to take over a lot of things that we can and should do in our build process.

That said it does some things that no other build system does. It works to create CSS and Javascript bundles from the same configuration (assuming that it is properly configured), it will warn you if any of your bundles are over 250Kb which prevents the bundles from slowing down page loading.

I think it's too big a tool that tries to do too much but, again, it seems that everyone is using it so you may be called to use it whether you like it or not.

We will only concentrate on Javascript bunding, code spliting and shaking. I will not use Webpack for handing images and S/CSS because I don't think we should be converting it all to Javascript. Think about it, we load over 1MB of images into our pages, and a couple hundred KB of CSS and we will convert all those images to strings just so we can feed them to Javascript and then place them back into their correct locations? Thanks but no thanks... I would much rather use traditional strategies built around Gulp and it's ecosystem rather than split it into something completely different

Installing Webpack

```
# install webpack and dependencies
npm install  webpack \
babel-core \
babel-loader \
babel-preset-es2015 --save-dev

# build project structure
mkdir -p project/src
```

Create the following files and place them in the location indicated in the first line of each file. If it doesn't have a path, like `src/to/file.js` then it goes in the project folder.

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta<html lang="en">>
    <title>Hello webpack</title>
  </head>
  <body>
    <div id="root"></div>
    <script s<title>t/bundle.js"></script>
  </body>
</html>
</script>
```

We'll place the script in the app directory. This example is deliberately simple, It can be as complex as you need it to be.

```
// src/app.js
const root = document.querySelector('#root')
root'#root'TML = `<p>Hello webpack.</p>`
```

The final script of this build process is the Webpack configuration file. This file will transpile the app.js file into ES5 and then convert it into bundle.js for use in all browsers.

```
// webpack.config.js
const webpack = require('webpack')
const path = require('path')

const config = {
  context: path.resolve(__dirname, 'src'),
  entry: './app.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  },
```

```
  module: {
    rules: [{
      test: /\.jsx$/,
      include: path.resolve(__dirname, /\.jsx$/'src'),
      use: [{
        loader: 'babel-loader',
        options: {
          presets: [
            ['es2015', { modules: false }]
          ]
        }
      }]
    }]
  }
}

module.exports = config
```

In order to run the files we need to modify the project's `package.json` to add scripts that will run Webpack. There are ways to incorporate the tool into Gulp and Grunt buuild processes, that is left as an exercisse to the reader.

```
{
  "scripts": {
    "build": "webpack"
  },
}
```

One last thing to consider. The configuration presented above will bundle everything together and that may not always be desirable.

We can modify the Webpack configuration script to bundle all our vendor assets into separate chunks. The modified script looks like this:

```
var webpack = require('webpack');
var path = require('path');
```

```
module.exports = func'path'nv) {
    return {
        entry: {
            main: './index.js',
            vendor: 'moment'
        },
        output: {
            filena'./index.js'[chunkhash].js',
            path: path.resolve(__dirname, 'dist')
        },
        plugins: [
            new webpack.optimize.CommonsChunkPlugin({
                name: 'vendor' 'dist'// Specify the common bundle's name.
            })
        ]
    }
}
```

By providing multiple entry points we make it easier to use the `CommonsChunkPlugin` to bundle together our vendor (normally third party) libraries. In the example we're bundling moment but this may also be an array of our vendor dependencies.

> If you want more information, check the Webpack Guides.

Why do we need to convert everything to Javascript?

Granted, everyone has an opinion on this subject. Mine is that I don't think everything in JS is the answer to all our development issues.

Modules natively in browsers

Rollup and Webpack are popular because, until recently, there was no way to work with modules directly in web browsers. Even though ES6 is almost 2 years old we had no browser that supported modules natively.

> That is changing. Native support for modules is starting to come out in

browsers... it's still experimental in most of them but it's a good sign that they will come into release version of the browsers soon. The current list of supported browsers:

- Safari 10.1.
- Chrome Canary 60 – behind the Experimental Web Platform flag in `chrome:flags`
- Firefox 54 – behind the `dom.moduleScripts.enabled` setting in `about:config`
- Edge 15 – behind the Experimental JavaScript Features setting in `about:flags`

Differences between regular scripts and module scripts when used in the browsers (taken from exploring ES6:

	Scripts	Modules
HTML element	`<script>`	`<script type="module">`
Default mode	non-strict	strict
Top-level variables are	global	local to module
Value of `this` at top level	`window`	`undefined`
Executed	synchronously	asynchronously
Declarative imports (`import` statement)	no	yes
Programmatic imports (Promise-based API)	yes	yes
File extension	`.js`	`.js`

There are two ways to create a module. External and internal modules, each of which can export and import multiple named functions from other modules.

Take the following `utils.js` external module that explorts a single function to append text to the body of a page.

```
// utils.js
export function addTextToBody(text) {
```

```
  const div = document.createElement('div');
  div.textContent = text;
  document.body.appendChild(div);
}
```

This internal module imports `addTextToBody` from `utils.js` and uses it as a local function without name spacing.

```
<script type="module">
  import {addTextToBody} from './utils.js';

  addTextToBody('Modules are pretty cool.');
</script>
```

The last concern when working with native module implementations is how to handle older browsers. Most modern browsers have repurposed the `type` attribute of the `script` element: If it's value is module the JS engine will treat the content as a module with different rules than those for normal scripts.

To target older browsers use the `nomodule` attribute.

```
<script type="module" src="module.js"></script>
<script nomodule src="fallback.js"></script>
```

There is a lot more to learn about native module implementation. Some of the things that caught my attention when researching modules:

They only run once per page no matter how many times you load it.

Modules and script modules never block rendering. They always run as if the `defered` attribute was set in the calling script tag. The defer tag means that the script will execute after the content is downloaded but before the DOMContentLoaded event is fired.

Modules and inline (script) modules can use async attribute meaning that they can be made to load without blocking HTML rendering but it also means that you can no longer guarantee execution order. If the order your scripts run in is

important rely on the defered attribute discussed earlier.

Must use a valid Javascript Mime Type or it will not execute. In this context, a valid Javascript Mime Type is one of those listed in the HTML Standard:

- application/ecmascript
- application/javascript
- application/x-ecmascript
- application/x-javascript
- text/ecmascript
- text/javascript
- text/javascript1.0
- text/javascript1.1
- text/javascript1.2
- text/javascript1.3
- text/javascript1.4
- text/javascript1.5
- text/jscript
- text/livescript
- text/x-ecmascript
- text/x-javascript

Links and resources

- https://jakearchibald.com/2017/es-modules-in-browsers/
- https://medium.com/@samthor/es6-modules-in-chrome-canary-m60-ba588dfb8ab7
- http://2ality.com/2014/09/es6-modules-final.html
- https://ponyfoo.com/articles/es6-modules-in-depth