Modules

WE'LL COVER THE FOLLOWING

Before we get started looking at Modules, we need to change the Gulp and Babel files that we set up earlier to allow us to work with modules. We need to add four new modules: `browserify`, `babelify`, `vinyl-source-stream`, and `vinyl-buffer`.

Support for Modules in browsers does not quite exist in the browser. Even though ES6 introduces modules in the language so we need to use something to transpile and bundle them. There are many popular module bundler's out there, in this book we will look at Browserify.

Browserify is a popular module loader. It allows you to take npm modules, or our own modules, and bundle them up into a single file! It takes the `require` functions from node and, as they say "recursively analyze all our dependences" in order and serve up a single file.

However if we are going to use the ES6 `import` and `export` keywords we need to first transform our code. Enter Babelify, this is a transformer for Browserify that will take our ES6 and make it work well together.

The `vinyl-source-stream` and `vinyl-buffer` modules are used to take the output from Browserify and gather it all up so that we can put it all in one file. The output is a stream, and we can used these to property pull everything together.

To get started make sure you have read the Using ES6 now chapter, as we will have a similar set up. You can also find the files for this and the previous

example on GitHub at https://github.com/lets-learn.

Make sure you navigate towards a folder you would like work in, to get started let's install our npm modules.

```
npm install --save-dev browserify babelify vinyl-buffer vinyl-source-stream
```

With these installed let's start changing the gulp file from the previous chapter.

```
const gulp = require('gulp');
const babelify = require('babelify');
const browserify = require('browserify')
const source = require('vinyl-source-stream');
const buffer = require('vinyl-buffer');
```

In our ES6 task we need to remove the current code and change it to look like this.

```
gulp.task('es6', () => {
  browserify('src/app.js')
    .transform('babelify', {
    presets: ['es2015']
  })
    .bundle()
    .pipe(source('app.js'))
    .pipe(buffer())
    .pipe(gulp.dest('build/'));
});
```

We now use `browserify` to take our source file, look for an modules in there, and get them! We transform it with `babelify`. This step is very similar to our original gulp file, where we set the preset to be `es2015`. Remember that Babel will not transpile our code from ES6 by default, so we need to make sure we include the `babel-preset-es2015`.

Next we need to bundle up our files. Here we call the `.bundle` method and pass that into our `vinyl-source-stream`. The `.bundle` method returns a stream so we use `vinyl-source-stream` to handle that stream, and `vinyl-buffer` takes that stream and creates an output file for us.

Export

Let's create our first module, there are a couple ways to create modules. Modules can have a single `default` export, or a module can have multiple named exports.

Start by creating a file called `myFirstModule.js`.

```
export default function(text) {
  console.log(`My first module is ${text}`);
};
```

We have two keywords here, `export` and `default`. `export` is needed to define that we are going to be exporting something, here we are setting up the `default` exports. A file can have multiple exports, but we can also specify a `default` to act as the fall-back.

Now that we have a simple default export set up, let's look at how we can import it. Create a new file called `app.js` and also create an `index.html` file to load our script. In the `app.js` file we can import our new module like this.

```
import React from 'react';
import first from './myFirstModule';

console.log(first('Neat!'));
```

In the `app.js` file we use the `import` keyword to declare our import, we then give it a name, in this case we are calling it `first`. We could name this anything we like. The `from` keyword is used to set the modules source location, in our case we are using a file in our directory.

With Gulp running, open that file in the browser, and you should see your text in the console.

Notice how we can leave the `.js` off, it is assumed that it will be a `.js` file, so we can exclude the extension.

We can also import npm modules, say we installed `react` via npm. We could import it like such.

```
import React from 'react';
```

Named exports

A file can only have one default export, but it can have many named exports. To create a named export we leave the `default` keyword off and supply a function or variable with a name. Create a new file called `myMathModule.js`;

```
export function add(a,b) {
  return a + b;
};
```

One way to import this named export is to use destructoring to get it. When we have named exports our modules now exports an object. Using destructoring we can pick the properties we want. If you need a refreshing on destructoring check out the destructoring chapter.

```
import { add } from './myMathModule';

const total = add(4,5);

console.log(total);
```

Let's add another export to see how this works.

```
export function add(a,b) {
  return a + b;
};

export function subtract(a,b) {
  return a - b;
};
```

Above we have added a `subtract` function, now if we wanted to import this in another file we can do this.

```
import { add, subtract } from './myMathModule';
```

This is great because we don't have to include all the modules if we don't need to use them. And tools like Rollup, which performs a method called tree shaking, will see this on only bundle the code used, not the entire module.

We can also pull in all of our named exports using the `*` character and the `as` keyword.

```
import * as myMath from './myMathModule';

myMath.add(2,3); //5
myMath.subtract(10,7); //3
```

One other thing to note about a module is that it can have named exports along with a default export. Let's change our `myMathModule` to include a default as well.

```
export function add(a,b) {
  return a + b;
};

export function subtract(a,b) {
  return a - b;
};

export default function (a,b) {
  return a / b;
};
```

Here we added a default export that acts as our divide function, now we can import this and our `add` by defining the default first, followed by a `,` and then the destructored `add`.

```
import divide, { add } from './myMathModule';
```

Modules are a great way for developers to break up their code into more manageable and organized pieces. They allow us to create maintainable code that is less daunting to get work with.

```
export function add(a,b) {
  return a + b;
};

export function subtract(a,b) {
  return a - b;
};

export default function (a,b) {
  return a / b;
};
```