Transpiling

Babel

You know the basics of ES5, NodeJS, NPM, and Webpack. All that's left is setting up a transpiler to allow writing ES6 code that can be run in the vast majority of browsers.

Babel, according to its homepage, is a "Javascript Compiler". In reality it's a *transpiler* that converts modern Javascript into broadly supported Javascript. There are a number of ways it can be executed, but for this session, we'll focus on hooking it into Webpack.

First, well need to install all the dependencies. Typically, when defining build-related dependencies like Webpack and Babel, you should define them in the package.json as `devDependencies`, not `dependencies`. The difference is that `devDependencies` will only be installed when running `npm install` directly in the component, and not when including the package as a separate project.

To use Babel in Webpack, you'll need to install three Babel components: the "loader", the "core", and the "preset". We'll install a specific version of Webpack (4.29.6), and any version of the necessary Babel components that are compatible with the versions specified.

```
{
  "name": "babel-sample",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/core": "^7.3.4",
    "@babel/preset-env": "^7.3.4",
    "babel-loader": "^8.0.5",
    "webpack": "4.29.6"
  }
}
```

Once they're specified, start the install.

While install is going on, I'll explain how the caret works. A caret operator in the version string of `devDependencies` means that when a developer installs or updates npm dependencies via `npm install` or `npm update`, any component version that does not introduce breaking changes will be permitted to update. So for example if the `@babel/core` dependency updated to 7.4.0, that version would be installed, but if it updated to 8.0.0, it would not update. The idea is to allow updating bug fixes or perhaps performance improvements the dependency's author introduces but not to install major breaking changes. This can be a little risky if you can't trust the dependency's author to honor semver conventions. For example, an author might make a breaking change but only release it as a "bugfix" update, thus mysteriously breaking your project.

So, here's what we just installed:

- `@babel/core` is the foundation of Babel and is what performs the transpiling operation.
- `@babel/preset-env` defines *what* needs to be transpiled. It offers options to allow developers to dictate their target browsers they need to support to utilize just the right amount of native ES6.
- `babel-loader` is what allows Webpack to make use of Babel.

Before going forward, let's create a front-end script, build script and sample page that does not make use of babel. I'll write the script planned for the front-end with a hint of mundane ES6:

`index.js`:

```
const salutation = 'Hello'

console.log(salutation)
```

The `const` keyword is a new variable type introduced in ES6. It signifies a block-scoped variable that's reference cannot be changed. It's useful when you want to ensure that the variable will not mutate later in the code.

Now, we'll write the build script that turns `index.js` into a bundle:

`build.js`:

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

webpack({
  entry: './index.js',
  mode: 'development',
  output: {
    path: path.resolve('./'),
    filename: 'bundle.js'
  }
}, function (error, stats) {
  if (error) {
    console.error(error)
  } else {
    console.log(stats.toString())
  }
})
```

Next, have this build script invoked from package.json:

`package.json`:

```
{
  "name": "babel-sample",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "build": "node build.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/core": "^7.3.4",
    "@babel/preset-env": "^7.3.4",
    "babel-loader": "^8.0.5",
    "webpack": "4.29.6"
  }
}
```

Next, create the index.html page that references this bundle:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Babel Sample</title>
  </head>
  <body>
    <script src="./bundle.js"></script>
  </body>
</html>
```

Now we just run the build script and check the output:

```
npm run build
```

Here's the output:

```
/***/ "./index.js":
/*!******************!*\
  !*** ./index.js ***!
  \******************/
/*! no static exports found */
/***/ (function(module, exports) {

eval("const salutation = 'Hello'\n\nconsole.log(salutation)\n\n\n//#
sourceURL=webpack:///./index.js?");

/***/ })
```

So the bundle generation is working, and the code more or less is unchanged: `const` is still present in the result.

If we open this in Chrome, it should work fine, as modern browsers have supported the `const` keyword for awhile now. But this isn't true for all ES6 (or even newer) features.

Now let's add some Babel options to the Webpack config:

`build.js`:

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

webpack({
  entry: './index.js',
  mode: 'development',
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env']
        }
      }
    ]
  },
  output: {
    path: path.resolve('./'),
    filename: 'bundle.js'
  }
}, function (error, stats) {
  if (error) {
    console.error(error)
  } else {
    console.log(stats.toString())
  }
})
```

Babel plugs in to Webpack via a "module rule". So we'll create a `module` object that contains an array of `rules`.

Inside the array of `rules`, we'll create an object that, given any file that ends in .js and is not in the node_modules folder, applies the Webpack `babel-loader`.

`test` is a regular expression that tests to ensure the extension is .js.

`exclude` is a regular expression that states to ignore any path that includes node_modules.

`loader` is the name of the loader to apply.

`options` define options specific to the loader specified. The babel loader needs a preset, or a set of plugins to support specific language features, to know what needs to be transformed. The default `@babel/preset-env` preset will tell the babel core to make enough transformations to support the vast majority of browsers.

Now, run build again:

```
npm run build
```

And now check out the bundle.js output:

```
/***/ "./index.js":
/*!*****************!*\
  !*** ./index.js ***!
  \*****************/
/*! no static exports found */
/***/ (function(module, exports) {

eval("var salutation = 'Hello';\nconsole.log(salutation);\n\n//#
sourceURL=webpack:///./index.js?");

/***/ })
```

Notice how `const` has been turned into `var`. Thus, we know the babel transform was successful! We can finally move on to developing ES6 for the browser without worrying about breaking older browsers!

Before finishing up, it's important to note that, just like Webpack, there are options for how to configure Babel, and different projects might make different approaches. Some projects may make use of the `.babelrc` file in the project root. Another option is to specify a configuration in a `babel.config.js` file. I prefer specifying the Babel configuration with the Webpack configuration in a dedicated script because it's explicit and isolated, as opposed to implicit and dispersed. But in the end it's up to what you or your team wants to do. Just make sure to not mix babel configuration options, and remember that some Operating Systems hide files that start with dots!

Well, we took quite the detour, learning how to program in NodeJS and how to use Webpack and Babel, but now that we're able to transpile modern Javascript into something all browsers can support, we're now ready to move on to learning ES6!

Congratulations for hanging on for this long. I can promise you that the projects going forward will be a lot more fun, and not just because you now have a really solid foundation.

With that, let's move on to ES6.