Tooling Lab

In this lab, you're going to get hands-on experience in setting up a development environment manually and from scratch.

Let's say that our team has decided to deploy our web application to a server as a transpiled, linted, minified, bundled app. To transpile it, we'll run it through Babel. We'll lint it with esLint. And we'll minify and bundle it using webpack. Finally, to manage all of this, we'll automate it using npm.

Creating a project from scratch

We'll use npm to manage our project and package. Let's start by initializing the project and then installing some tools.

1. Open a bash window and cd to your Labs directory.
2. Run this command:

```
npm init
```

3. Answer the questions after discussing each with your partner. Let the entry point be "index.js" and the test command be "jasmine".
4. Open package.json file in an editor. Look at what you've created. What format is it in? _____ What are some of the keys and values? _____

Now let's install some tools we'll need to develop and serve web pages on a bona-fide web server.

5. Run this command:

```
npm install --save-dev webpack webpack-cli
```

6. Look at package.json again and discuss with your partner how it has changed.
7. Also look at the node_modules folder. Examine the structure briefly.

Linting the code

We'll use eslint to perform static code analysis each time we build. Let's start by installing it. Then we'll create the .eslintrc.js configuration file using eslint itself.

8. Install eslint:

```
npm install --save-dev eslint
```

9. Run eslint for the first time like so:

```
./node_modules/eslint/bin/eslint.js --init
```

   Tell it you want to answer questions. You and your partner decide how you think you should answer them. Take your best guess at each but don't get stressed; you're just creating a config file that can be changed later if you don't like an answer. (Some hints, though, ES2018, yes to modules, JavaScript config file).
10. Once you're finished answering questions, take a look at .eslintrc.js. Again, discuss with your partner about the format and contents.
11. Lint your code like so:

```
./node_modules/eslint/bin/eslint.js . --ext .js
```

12. Unless your code is perfect, the linter will have made some suggestions. When suggestions are made, you should A) take their suggestion, B) change the rule, or C) create an exception. You and your partner should discuss what to do for each problem reported by the linter.
    - To take their suggestion, edit your JavaScript code.
    - To change the rule, edit .eslintrc.js and modify or delete the rule you don't like.
    - To create an exception, turn off a specific rule for a specific file. Put a line like this at the top of the file:

```
/* eslint ruleNameHere:settingHere */
```
Or this to ignore the rule on a single line
```
someJavaScriptCommand  // eslint-disable-line ruleNameHere
```

Once your problems are resolved, let's move on!

Bundling and minifying it

13. Remember, we'll need webpack for this part and webpack requires a config file called webpack.config.js. Go ahead and create webpack.config.js in your Lab folder with this in it:

```
const path = require('path');
module.exports = {
  mode: "development",
  entry: "./src/index.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename:"bundle.js" },
  module: { rules: [] },
  plugins: [],
}
```

14. Run webpack from your bash window by typing

```
./node_modules/webpack/bin/webpack.js
```

15. Look in the dist folder for bundle.js. Examine it. See how much simpler they are than the original? (Kidding).
16. Notice that your index.html file isn't pointing to your bundle. We could fix this by hand but it wouldn't be maintainable. Let's fix that next with a webpack plugin.

A webpack plugin

17. Install the HTML webpack plugin:

```
npm install --save-dev html-webpack-plugin
```

18. Import the library at the top of webpack.config.js:

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
```

19. Register the plugin:

```
plugins: [new HtmlWebpackPlugin()],
```

20. Run webpack again and look in the dist directory. You should see a file called index.html. But it isn't your index.html. Let's fix that next.

```
plugins: [new HtmlWebpackPlugin({template:'src/index.html'})],
```

    This says to take your exiting index.html file and use it as the template for the new index.html file.
21. Run webpack again and look at your new index.html file. You'll see that it has two <script> tags, one we hardcoded originally and one added by webpack.
22. Edit your original one and remove your script tag.
23. Run it again and verify that the new index file has only one <script> tag -- the one added by webpack.

webpack loaders

Let's see how webpack allows us to style our page with css.

24. While it is running, look at the font. It is currently the default font which is a serif font.
25. We have a CSS file that you can use in the starters folder. It is called *site.css*. Copy it to the src folder.
26. Install the CSS and style loaders:

```
npm install css-loader style-loader --save-dev
```

27. Register them with webpack by putting them as a ruleset in the module section of webpack.config.js:

```
module: {
  rules: [
    { test: /\.css$/,
```

```
      use: [{ loader: 'style-loader'},{ loader: 'css-loader'}]
    },
  ]
},
```

28. Lastly we must signal webpack that this resource is a dependency. We can do that by importing it. Add this to index.js:

```
import './site.css'
```

29. Re-run webpack and browse to your page. If your font is now a sans-serif font, it worked!

Transpiling the code

We would really only **need** to transpile if we were using very new JavaScript features and had customers who use very old browsers. But let's say that were the case. Let's get some experience with transpiling with Babel. First, let's use some new features of JavaScript.

30. Look in the setup folder for a JavaScript file called getBrowserInfo.js. Copy this file into your src directory. Examine the file. It uses a lot of ES2015+ syntax.
31. Let's import it into your other JavaScript file. Put this at the top:

```
import getBrowserInfo from './getBrowserInfo';
```

32. And put this at the bottom:

```
document.getElementById("main").innerHTML = getBrowserInfo();
```

33. Run webpack and look at the bundle.js in the dist directory. Do you see the minified code from getBrowserInfo.js? You should.

Now let's transpile it.

34. Install Babel:

```
npm install --save-dev babel-core babel-preset-env babel-loader
```

35. Babel uses a config file called .babelrc. Go ahead and create that now:

```
{
  "presets": [ "env" ]
}
```

36. Tell webpack to transpile your code using Babel by adding a new rule to the module section of webpack.config.js (hint: you'll list it before or after your css loaders. Make them comma-separated):

```
module: {
  rules: [
    { test: /\.css$/, use: [ ...]},
    { test: /\.js$/, use: { loader: 'babel-loader'}}
  ]
},
```

37. Run webpack again and refresh your page. You should see the page now has a bunch of info about your current browser and OS.
38. Take a look again at the minified code in bundle.js. Compare it with the original code.

Automating the build with npm

We're using npm for two things so far; to hold metadata about our project and to manage our packages/libraries. Let's use it for a third thing; to automate processes.

39. This command should fail but run it anyway:

```
npm run build
```

40. Once again, edit package.json. Remind yourself that it has a "scripts" section. This is made for automation. It's an object with the command on the left and the backing expression that will run on the right. We'll start by adding a new script.
41. Add a new entry like this:

```
"build": "eslint src --ext .js && webpack"
```

42. Save the file and do this again:

```
npm run build
```

43. This time it should work great and will probably expose some new linting warnings or errors.
44. Go ahead and try to resolve the linting output until your project builds without errors or warnings.