Development Environment

Let's get your development environment up and running so that we can start writing some code shall we?

Installing Node

Installing Node is fairly straight forward on Mac OSX and Windows. The NodeJS website provides installers for both platforms. Just click on the `v5.3.0 Stable` button and you'll be off and running.

If you're using Linux, there isn't a premade installer, but you can download the binaries from the downloads page. Just be sure to click on the `Stable` button so that you're installing the latest and greatest version of NodeJS. If you don't know how to install from the binary on your Linux system, there's a great answer for this on StackOverflow.

Installing npm

You're done already. npm is installed alongside Node.

Installing React

Next we're going to go ahead and get React installed. As of React 0.14, it is broken up into several modules. `React` and `ReactDOM` are the two major ones, and what we will start with.

But first we need to create a folder for our app so we have a place to store all of this stuff. Go into whatever directory you like to store all of your code in and make a new directory inside of that one:

```
$ mkdir react-reading-time
```

By the way, we'll be making a small reusable React component in this tutorial that will show estimated reading time for an article. Neat eh?

Now let's `cd` into that directory and create a boilerplate `package.json` file:

```
$ cd react-reading-time
$ npm init --yes
```

Using the `--yes` flag just skips all prompts and uses the defaults to initialize the `package.json` file.

After this is done we can install `React` and `ReactDOM`:

```
$ npm install --save-dev react react-dom
```

You'll notice that we added the `--save-dev` flag. This tells npm to add both of those packages to the `devDependencies` section of our `package.json` file. We're saving it as a development dependency because this little module will likely be a part of a bigger application that will already have React installed.

Great! React is now installed and all we need to do now is install a few more packages and get our development server up and running!

Development server

In order to get our development server up and running we're going to have to install a few modules, and create a couple of new files. Let's start by installing a few things.

Webpack

We're going to use Webpack to pipe our ES6 code through Babel and bundle the ES5 output for optimal download and execution speed. Webpack also provides a development server.

With the development server our assets will be `hot-reloading`, which means any changes we make to our included JavaScript or CSS files will be automatically compiled, updated in the bundle, and injected into the existing browser tab without us having to reload to see our changes. Now THAT is awesome stuff.

Let's begin by installing Webpack. We will be installing Webpack globally so that we have access to the Command Line tools. We do that by using the `-g` flag:

```
$ npm install -g webpack
```

Install Global What's this `install global` stuff mean anyways? Up until now we've been installing all of our packages locally, meaning that npm fetches the modules, and installs them in the `/node_modules` directory in the root of our app. This is great for modules that *only* our app is dependent on, or modules that don't have a CLI, but for things like Webpack or Grunt this is necessary.

Babel

Ok now that we have that all cleared up let's move on and install all of the Babel dependencies we will need:

```
$ npm install --save-dev babel babel-core babel-loader
```

We'll also need to install the babel preset modules so it knows how to transpile all of our code:

```
$ npm install -save-dev babel-preset-es2015 babel-preset-react babel-preset-stage-0
```

Great! Babel and all of the dependencies related to Babel have been installed. But wait a minute! At the end of the install npm complained in big bright red letters: `UNMET PEER DEPENDENCY webpack@^1.0.0`. But we just installed Webpack didn't we? We did! But if you'll remember we installed it globally. When npm was resolving peer dependencies it detected that we didn't have Webpack installed anywhere in our `node_modules` directory, and we don't, so let's install it locally now.

```
$ npm install --save-dev webpack
```

Peer Dependencies Wait up a second, now what's a peer dependency? Simply stated, it's a module that has code that is needed for the parent module to run properly, but shouldn't necessarily be included with the module. For instance, if you're writing a reusable React component that you would like to publish on npm for the whole world to use, it's not really a great idea to include React as a dependency. It's inferred that the person using this module is already going to have React installed in their project, and therefore not necessary to include in the dependency tree.

Webpack Dev Server and React Hot Loader

These are the two pieces that allow us to do the hot-reloading that we were talking about earlier. Let's go ahead and install them:

```
$ npm install --save-dev webpack-dev-server react-hot-loader
```

Configure the server

In order to configure the server we're going to have to create a configuration file for Webpack. This isn't a Webpack class, so I won't go into the details. First create a directory for our example application, and create the config file:

```
$ mkdir example
$ cd example
$ touch webpack.config.js
```

Now open up that file and copy the following code in:

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

module.exports = {
  entry: {
    'react-reading-time': [
      'webpack-dev-server/client?http://localhost:8881/',
      'webpack/hot/only-dev-server',
      './example/react-reading-time.jsx'
    ]
  },
  output: {
    path: __dirname,
```

```
    filename: "[name].js",
    publicPath: 'http://localhost:8881/',
    chunkFilename: '[id].chunk.js',
    sourceMapFilename: '[name].map'
  },
  resolve: {
    extensions: ['', '.js', '.jsx', '.es6'],
    modulesDirectories: ['node_modules']
  },
  module: {
    loaders: [
      { test: /\.jsx$|\.es6$|\.js$/, loaders: ['react-hot', 'babel-loader'], exclude: /node_module
      { test: /\.scss$|\.css$/, loader: 'style-loader!style!css!sass' }
    ]
  },
  plugins: [
    new webpack.NoErrorsPlugin()
  ],
  devtool: "eval-source-map"
};
```

Awesome! We've got our Webpack configuration in place. Now let's add some scripting commands to our `package.json` file so we can easily start it up. Open up `package.json` and add the following line in the `scripts` object:

```
"start": "webpack-dev-server --config ./example/webpack.config.js --hot --port 8881"
```

We won't be doing any testing in this tutorial, so go ahead and remove the "test" line. Your `scripts` object should now look like this:

```
"scripts": {
  "start": "webpack-dev-server --config ./example/webpack.config.js --hot --port 8881"
}
```

This will give us the `npm start` command which we can use to fire up our development server.

Now we can fire up our development server. Let's give it a shot:

```
$ npm start
```

You should see Webpack initialize and try to compile our bundle. It's going to complain and tell you that it can't find `example/react-reading-time.jsx`, and that's ok because we haven't created that file yet! We'll be doing that shortly!

You can also visit `localhost:8881` and see the glorious web server in action. Although there isn't much to see just yet...