Using ES6 now

WE'LL COVER THE FOLLOWING

While support for ES6 is increasing for evergreen browsers such as Chrome and Firefox, we can't always assume that users will be on one of those browsers. So, in order to utilize ES6 features now and make sure we won't run into cross-browser compatibility issues, we need to transpile our code.

Let's look at two possible ways we can perform the task of transpiling our code. First, we will use npm scripts and babel, and for the second, we will look at using gulp with babel.

Babel

Babel (1) is the go-to transpiler for ES6. It was originally called 6to5, but was later renamed to Babel as it was apparent that the name would not work moving forward (2). With the release of Babel 6, the focus turned more towards making Babel pluggable. It created a system that allows you to create plugins to transform your code! The default action for Babel in version 6 is not

to transform your ES6 to ES5 anymore, so you now have to include various presets.

The presets in Babel allow you to either pick and choose the transformations that you want, or you can select the `babel-preset-es2015` preset and work with all the features.

Babel CLI

In order to work with the two methods that we will look at, it is important to make sure you have Node.js installed. The easiest method would be to head over to the Node.js website and download the latest version for your operating system.

If everything goes as planned, you should have `node` available to you in your terminal. To confirm that Node.js is installed on your system, open your terminal and type `node -v` into the prompt.

```
$ node -v
v5.2.0
```

If you get a response that looks something like what you see above, you are good to go! If you are not super comfortable with the command line, I suggest checking out commandlinepoweruser.com (3) by Wes Bos. It's a great free video series to help you quickly get up and running with the common commands.

Once we have Node up and running we need to install the Babel CLI. To do so we will use npm. To test this, create a project folder and navigate there in the terminal. To get started we need to create a `package.json` file. To get this started with npm, we can run:

```
npm init
```

This will walk you through a few questions: what is the name of your project, version, description, etc. When it asks you about the `entry point`, you can leave it as `index.js` for now. Typically this would be the main file for your project. If you know ahead of time what that will be, add it there.

Once you are done with these steps, a new `package.json` file will have been created. Now, it's time to install the babel package we need.

```
npm install --save-dev babel-cli
```

After running this line you will see a few new changes in your project directory. First, you will notice that there is now a `node_modules` folder, and second, that there is a new entry inside your `package.json` file. There is one key part to the code above: the `--save-dev` flag. This flag will save the module, `babel-cli`, in your package.json file under `devDependencies` section.

```
"devDependencies": {
  "babel-cli": "^6.6.5"
},
```

This section in the file is used for modules that will be used in development, meaning they are not needed in production. If we had run our install with `--save` instead, it would put the module under the `dependencies` section.

But, what is the point of all these sections? If you are unfamiliar with npm and the `package.json` file, it's a file that you can take from project to project and run `npm install` and anything listed in `devDependencies` or `dependencies` will be installed for that project. If you wanted to install just the `dependencies` you could run:

```
npm install --production
```

There will be cases when you are developing your apps that you will want to use certain modules exclusively for development, and certain modules exclusively for production. Using `--save` and `--save-dev` you can put those modules in the right buckets.

Babel Presets

As mentioned earlier, Babel does not default to transpiling ES6 right off the bat. We have to include the `babel-preset-es2015` module as well.

```
npm install --save-dev babel-preset-es2015
```

This module ensures that we have the ability to transpile our code.

Folder structure

Before we go too far, lets talk about folder structure. If you are going to be following along, I have set up my folder structure as follows:

```
- project
    - src
      - app.js
    - public
    - package.json
```

In the root `project` folder I have a `src` folder and a `public` folder. The `src` folder is used to hold my pre-transpiled code. This would be the code that is written with the nice ES6 syntax, while the `public` folder will be the destination for our transpiled code. The `public` folder would also hold the CSS, HTML, and images that your app requires.

NPM Script

The first method of transpiling we will look at is using NPM to run Babel. In the `package.json` file there is a section called `scripts` where we can define commands we would like to run. Using the Babel CLI we can set up a script to compile our code and output it where we would like. The `scripts` section takes a JSON object as a value. The keys in this object will be the name of our script and the value will be the command to run.

```
"scripts": {
  "js" : "babel src/app.js --out-file public/app.js"
}
```

Adding the above to our `package.json` in the `scripts` section will take our code from `src/app.js`, run it through Babel, and output the file in `public/app.js`. The Babel command has many options to it that we will look at later, but the `--out-file` flag is used to determine the destination of the compiled source.

Go ahead and add some ES6 code to the `src/app.js` file. If you don’t know what to add, try this:

```
const person = {
  name: 'Ryan Christiani'
};

const testString = `My name is ${person.name}`;
```

If you type `npm run js` into your terminal you should have an output of something like this:

```
> testfiles@1.0.0 js /Users/yourusername/path/to/project
> babel ./src/app.js --out-file ./public/app.js
```

Now, the moment of truth: let’s look in the `public/app.js` file. Note that if you did not create one before, Babel will go ahead and create it for you.

```
const person = {
  name: 'Ryan Christiani'
};

const testString = `My name is ${person.name}`;
```

Huh… this looks exactly the same. That is because we have not utilized the `es2015` preset yet!

babelrc

In order to tell Babel that we want to use presets that we have downloaded, we need to create a configuration file. This file is called `.babelrc`. If you have never worked with a dot-file before, you might think that it is a bit odd! Dot-files are meant to be hidden files and are typically used to store configuration information, or just to keep things hidden. By default you cannot see dot-files on your computer, however a quick Google search will show you how to turn them on for your finder. Since we are on the terminal, we can use the `ls -a` command to show any file or folder that starts with a `.`.

Sublime Text (the editor I am using throughout this book) or Atom will show you these files if you open your project folder in the editor.

In order to configure Babel to use our preset, we need to add this code to the `.babelrc` file.

```
{
  "presets": ["es2015"]
}
```

Babel configuration files take a JSON object with a `presets` key that contains an array of the presets you would like to use. There are many presets you can use, check out the Babel docs [4] for more information. Now, if you run `npm run js` it will properly transpile our code, changing the output to look like this:

```
'use strict';

var person = {
  name: 'Ryan Christiani'
};

var testString = 'My name is ' + person.name;
```

This is great, however let's make our script a little more robust. Right now, when we run the script, it will output the file and stop. If you want to keep working on the file we need to run this over and over again. We can configure the script to watch over the files and run when they change!

```
"scripts": {
  "js": "babel ./src/app.js --watch --out-file ./public/app.js"
}
```

By adding the `--watch` flag, we are telling Babel to listen for any changes in the `src/app.js` file, and when changes are made we want to output a transpiled version of that code to `public/app.js`. There are plenty more things you can do with npm scripts and the Babel CLI [5], however let's see how we can use Gulp [6] and Babel to transpile our code!

Gulp

Gulp is a popular task runner that helps to automate your workflow. There are a ton of different task runners out there, including Grunt, Broccoli, and Brunch. For this book, we will look at how to configure Gulp to automate working with Babel, and later on in the book, we will use this to work with modules as well. So lets get started!

Getting started with Gulp

In order to get started with Gulp, we need to install it globally on our system. Gulp is another npm module, so we install it like such: `npm install --global gulp-cli`. When working with gulp we have to install it globally only once, and also locally in each new project. We can do this by running: `npm install --save-dev gulp`.

gulpfile.js

To configure gulp we must first create a new file called `gulpfile.js`; this will be used to define our tasks. If you have never worked in Node before, you might not be familiar with the `require` function. This is a function that will pull in any modules from the `node_modules` folder, or your own modules. Since we ran `npm install --save-dev gulp` that placed it in our `node_modules` folder. In the `gulpfile.js` we can get access to `gulp` by requiring it.

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

This will store Gulp on a new variable called `gulp`. You will notice that I used `const` here! This `gulpfile.js` will be our first JavaScript file that uses ES6. I won't be going into much detail on the ES6 features we use here, that's what this book is for! We will discuss these features in more depth later throughout this book.

Gulp is all about tasks. We will start by defining some simple tasks in this file.

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

gulp.task('js', () => {

});

gulp.task('default', () => {

});
```

The `() =>` above is the syntax for an ES6 arrow function. Don't worry if you aren't familiar with this - we will discuss it later.

The two tasks we have created will work together, and also separately if we would like. The first of the tasks is one we will call `js` (note that you can call a task anything you want with the exception of `default`). The `js` task will deal

with taking our JS and running it through Babel. In order to run this task we can run `gulp js` in our terminal. Of course, we have to define the task first.

The second task is called `default`. This is a special task: in here we can set up code that will kick off many other tasks, or in our case create a listener for when certain files change. To run this task we can simply type `gulp` in the terminal. Gulp knows that if it just sees `gulp` it will run the `default` task.

The js task

Before we get started we have to install a module the will allow babel to work with gulp. Oddly enough this module is called `gulp-babel`. Type `npm install --save-dev gulp-babel` into your terminal to start downloading the `gulp-babel` package.

In our gulp file, let’s add these lines:

```
const gulp = require('gulp');
const babel = require('gulp-babel');

gulp.task('js', () => {
    return gulp.src('./src/app.js')
        .pipe(babel({
            presets: ['es2015']
        }))
        .pipe(gulp.dest('./public'));
});

gulp.task('default', () => {

});
```

The first thing we do is to require `gulp-babel`, then in our `js` task we return `gulp.src('./src/app.js')`. We return this to let gulp know that this will be an asynchronous event. The `.src` method lets us set the path for our file, and in this case it is a single file. When we get to the modules portion of this book we will change this to be a bit more robust. Gulp works with streams, meaning that we can keep moving the data down the pipe, so using the `.pipe` method we take the source files and send them through babel.

Just like earlier, we need to specify the presets we want to use. To do this, we can pass an object to the babel function to tell the babel plugin which preset we would like to use. Our source gets transformed into browser-ready code,

and it gets passed along again, so we can pipe it into `gulp.dest`, which sets the final destination.

Now that our task is complete, running `gulp js` in the terminal should produce this output!

```
[19:50:35] Using gulpfile ~/Sites/lets-learn/book/testfiles/gulpfile.js
[19:50:35] Starting 'js'...
[19:50:35] Finished 'js' after 102 ms
```

If you don't have any errors, then everything should be good. Gulp will have run, taken your `app.js` from the `src` folder, transformed it, and saved it to the `public` folder.

You might have noticed that this only ran once. If we change some code, we have to run it again. Let's set up the ability for gulp to keep watching our files.

```
gulp.task('default', ['js'],() => {
    gulp.watch('src/app.js',['js']);
});
```

Altering the `default` task we add an additional argument, an array, for the tasks that we want to fire when we first use the task. The `gulp.watch` method takes a few arguments: a file path that waits for file changes, and an array of the task(s) you want to run. Now we can just run `gulp`.

```
$ gulp
[21:26:50] Using gulpfile ~/Sites/lets-learn/book/testfiles/gulpfile.js
[21:26:50] Starting 'js'...
[21:26:50] Finished 'js' after 102 ms
[21:26:50] Starting 'default'...
[21:26:50] Finished 'default' after 6.73 ms
```

Changing your `src/app.js` file will run the `js` task again! Let's get into learning about ES6 so that we can use this file. In a later chapter we will add to this file.

Additional Resources

- 1:https://babeljs.io/

- 2:https://www.youtube.com/watch?v=rKuNbEwoQfQ

- 3:http://commandlinepoweruser.com/
- 4:http://babeljs.io/docs/plugins/#presets
- 5:https://babeljs.io/docs/usage/cli/
- 6:http://gulpjs.com/