Intro to Babel

Making the Latest Code Run on the Oldest Browsers

Remember That New Coolness we Learned?

- We've picked up some new cool stuff lately
  - let, const, arrow functions, array methods etc.
- But there are plenty of browsers that don't support them yet
  - IE8 doesn't have all of the array functions
  - IE11 doesn't support *any* of what we learned
  - Classes weren't available in Chrome / Firefox until 2016
  - Spread operators *still* don't work in Edge or Safari
- But we still want to write code using this new awesomeness!

Introducing Babel

- Named after the Tower of Babel, a biblical place where everyone spoke the same language, Babel translates our fancy javascript into the kind every browser can handle
- It was written by one 17 year old kid in 2015 to manage all of the emerging browser disparity, and is now downloaded 600,000+ times a month
- It does so by either converting newer style javascript to simpler versions of the same code, or creating "shims", plain functions that do what a browser should do internally
  - This is what is known as a "transpiler", a compiler that simply translates
- Let's see how it works using the Babel repl:

https://babeljs.io/repl/

Running Babel

- We can run this code locally by installing the npm package `babel-cli`
- This performs the same process as the one we saw online, only to local files
- Just download the module and add a package.json script

```
// package.json
{
    // ...
    "scripts": {
        "babel": "babel js/app.js --watch --out-file dist/app.js"
    },
    // ...
}
```

```
# In your terminal
npm install --save babel-cli
npm run babel
```

But Nothing Really Changed?

- Babel out of the box doesn't do any of the translation, only the interpreting
- We need to tell it what ruleset to use to transform the code
- We could define our own rules, or we could use a **preset** module
  - We'll use the `"latest"` and `"stage 0"` presets
  - `latest` gives us the features that are available in the latest browsers
  - `stage-0` gives us features that haven't *quite* made it into browsers yet
- We can configure this in a file at the root of our project called `.babelrc`

```
// .babelrc
{
    "presets": ["latest", "stage-0"]
}
```

```
# In your terminal
npm install --save babel-preset-latest babel-preset-stage-0
npm run babel
```

Wait, But What About Webpack?

- In the previous lesson, we setup webpack to be a pipeline for our files
- Rather than have a separate command for babel, why not have it be a part of the webpack build process?
- We'll do that by getting a babel **loader** and adding that to the config
  - Loaders are modules that manipulate files during the webpack build, we'll be adding more later
- Undo the previous `babel-cli` changes, and add the following

```
// webpack.config.js
module.exports = {
    // Previous config...
    module: {
        rules: [{
            test: /\.js$/,
            use: "babel-loader",
        }],
    },
};
```

Webpack Config Explained

- The `module` key defines configurations that involve other modules
- The `rules` key is an array of "rules" for what files get what loaders run on them
- Each rule has a `test` key that is a regular expression to run on file names to see if it should run on them
  - `/\.js$/` means files that end ($) in `.js`
- They also have a `use` key that says which loader(s) to apply to the files
- We'll be adding more of these in later lessons

Additional Reading

- Webpack Loaders Explained
- List of Common Loaders