Building

Modern JS web apps often have some level of "build" step(s)

- Transpiling JS dialects into native JS
- Transpiling new JS to older JS
- CSS Preprocessors/CSS Postprocessors
- Bundling multiple JS files for development into one JS file for deployment
- Minifying JS/CSS/HTML files
- Linting files to confirm syntax conventions
- Running unit tests to confirm functionality

Source Maps

- Transpiling/minifying creates "sourcemap" files
- Tells debuggers how to relate result to original

Starting a new package

```
mkdir demo
cd demo
npm init -y
npm install express
mkdir public
```

- Create `server.js` that loads static files from `public`

HTML Scaffolding

- Create static `index.html` and `page2.html`
    - `index.html`: "Page 1" in text
    - `page2.html`: "Page 2" in text
    - Has a form
        - `index.html`: `action="page2.html"`
        - `page2.html`: `action="index.html"`
        - Has a submit button
    - Loads `demo.js` and `demo.css`

Confirm setup

`node server.js` and visit `/` (for whatever PORT you configured)

- Confirm the two pages have forms that let you go from one to the other

Non-HTML Scaffolding

`demo.css`:

```
* Puts a border around `<form>`
```

`demo.js`:

```
      const sound = 'hi';
      console.log(sound);
```

```
* In an IIFE
* Add `use strict;`
```

Babel

Babel is the most common JS Transpiler

- Converts newer JS to older JS
- Converts JS dialects into vanilla JS
    - e.g. Typescript, JSX, future JS
- Allows for modern development without requiring user updates

https://babeljs.io

Copy your demo environment

Commands below based off of **https://babeljs.io/docs/en/usage**

```
mkdir demo-babel
cd demo-babel
npm init -y
npm install --save-dev @babel/core @babel/cli
npm install --save-dev @babel/preset-env core-js@3
```

`@babel` is npm's "namespacing" of packages.

Configure the Demo Environment

Create `babel.config.js` in package root

```
const presets = [ [
  "@babel/env",
  {
    targets: {
      edge: "17",
      firefox: "60",
      chrome: "67",
      safari: "11.1",
      ie: "9.0",
    },
    useBuiltIns: "usage",
    corejs: "3",
  },
] ];
module.exports = { presets };
```

Creating "source" files

The files you develop on are **input** to babel

- We will use `src/`
- Put `demo.js` into `src/` (**NOT** `public/`)

Babel will **output** the files for use

- We will use `public/
- Put `demo.css` into `public/`

What is the difference...

- Between css and JS files?
- Between `src/` and `public/`
- Which will be the webserver document root?

Running Babel

This command transpiles the files in the input directory (src) into the --out-dir (public)

- Configuration decides files and transformations

```
npx babel src --out-dir public
```

- Babel is often used with a bundler, like `webpack`
- Can write a `package.json` `scripts` entry for this
    - such as `npm run build`
- When using webpack, don't use babel by itself

Webpack

Webpack is a **bundler** - it pulls together multiple development files into one file for deployment

Webpack *also* can run other build steps. We can have it run babel on our files, for example

Copy your demo environment

Commands below based off of
https://webpack.js.org/guides/getting-started/

```
mkdir demo-webpack
cd demo-webpack
npm init -y
npm install --save-dev webpack webpack-cli
```

Configure webpack

Create `webpack.config.js` in package root

```
const path = require('path');
module.exports = {
  mode: 'development',
  entry: './src/demo.js',
  devtool: 'source-map',
  devServer: {
    contentBase: path.join(__dirname, 'public'),
    compress: true,
    port: 5000
  },
  output: {
    filename: 'demo.js',
    path: path.resolve(__dirname, 'public'),
  },
};
```

Create "source" files for webpack

The files you develop on are **input** to webpack

- We configured it to use `src/demo.js`

Webpack will **output** the files for us

- We configured it to use `public/demo.js`

```
mkdir src
```

Put `demo.js` into `src/`

Running Webpack

Run all the webpack steps (bundling, transpiling, etc)

- on the "entry" file (`src/demo.js`) and its imports
- generates the output (`public/demo.js`)
- config decides which files, what transformations

```
npx webpack
```

- Webpack is a lot of "magic" -
  - Do the steps by hand before relying on it!
- Often you write a package.json `scripts` entry
  - such as `npm run build`

Example exports - default

```
export default { one: 1, two: 2 }; // Some object, default
```

You can declare and initialize a variable on different lines than where you export it

```
const cat = { name: 'Maru' };
export default cat;
```

- The variable type (`var`/`let`/`const`) is true only for WITHIN the file.
- Once imported it is a new variable (even if it has the same name) with new rules.
- A file has at most one default export

Example imports - default

Webpack lets you use ES6 style "import/export" syntax

- instead of Node-style "require()/module.exports"

```
import theDefault from './module-a';
// let theDefault = require('./module-a');
```

- a default import gets the variable name you say
- most commonly: camelCase/MixedCase of name

```
import moduleA from './module-a';
```

Example exports - named

```
export const cat = 'Meow'; // exports named string
export const dog = ['drool', 'smell']; // exports named array
```

- You can declare and initialize a variable on different lines than where you export it
- `var`/`let`/`const` is true only for WITHIN the file
- Any import is a new variable with new rules
- Named exports/imports are same name by default
- A file can have any number of named exports

Example imports - named

```
import {namedOne, namedTwo} from './module-b';
// creates variables "namedOne" and "namedTwo"
```

- a named imported gets the same variable name as what it was exported as, unless you explicitly tell it a different one with "as"

```
import { namedOne as myVersion, namedTwo } from './module-b';
// creates variables "myVersion" and "namedTwo"
```

Example imports - collected

```
// module-a.js
export default { catLover: true };

// other-file.js
import theDefault from './module-a';
```

```
// module-b.js
export const namedOne = 'One';
export const namedTwo = 'Two';

// other-file.js
import {namedOne, namedTwo} from './module-b';
```

```
// module-c.js
export const namedOne = 'One';
export default namedThree = '4';

// other-file.js
import alsoDefault, {namedOne as other} from './module-c';
```

Webpack-dev-server

If ONLY static assets, can speed up DEVELOPMENT

- `npm install --save-dev webpack-dev-server`
- add a `devServer` section to your webpack.config.js

```
devServer: {
  contentBase: path.join(__dirname, 'public'),
  compress: true,
  port: 5000
},
```

- run `npx webpack-dev-server` not `npx webpack`

auto rebuilds AND the server auto shows the changes AND the browser auto reloads those changes

Using nodemon

`webpack-dev-server` is no help for dynamic assets

- `npm install --save-dev nodemon`
- `npx nodemon server.js` instead of `node server.js`
- Auto-restarts server
    - Does not auto update browser
    - extra configuration needed to do a rebuild of any assets (such as running webpack for you)