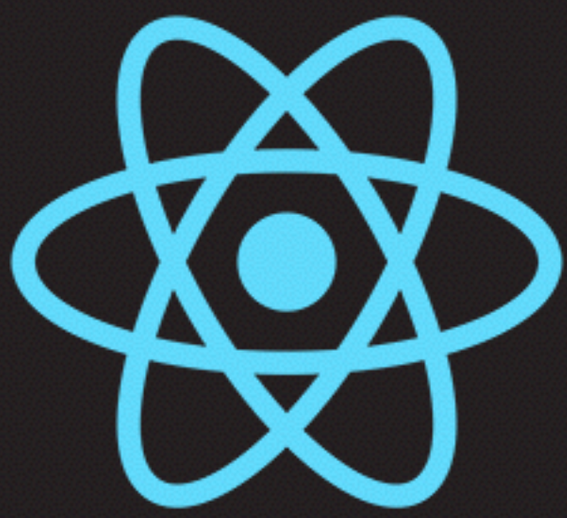
COLLEGIO TECHNOLOGIES INC.

WEBPACK, THIRD PARTY COMPONENTS, AND STYLING YOUR APP



WEBPACK INTRODUCTION

- So far, we've built out our apps using one main app.js file, and have been transpiling it by using the Babel.js command line tool. However, in the real world, this is an impractical solution.
  - Larger, more elegant applications require the ability to be broken down into separate files. At the very least, each component should be placed within it's own file.
  - But how do we tell Babel.js to transpile all of those files into valid JS? This is where Webpack comes into play.

- Webpack is known as a ***module bundler***. With it, we can perform multiple functions such as transpiling our codebase from ES6 and React into ES2015, converting SASS files to CSS, minifying files, and calling 3rd party modules when needed.

INSTALLING AND CONFIGURING WEBPACK

- From your command line, move into your project folder and run **yarn add webpack webpack-cli**. This will install Webpack in your project.

- In order to run Webpack, you will need to create a config file.
  - Web pack config files are stored in the root directory of your project and is called ***webpack.config.js***.
  - Technically, the webpack.config.js file is a Node.js script.
  - You can see the available config file options at https://webpack.js.org/concepts/

- Finally, add a script into yarn to run Webpack
  - **"scripts": {**
    "build": "webpack --watch"
    }

- Now that we have Webpack installed, we can ***import*** and ***export*** components and functions within our application. For more information on importing and exporting files in ES6, please see the following online resources:
  - **Import:** https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/import
  - **Export:** https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export
  - **Import/Export CheatSheet:** https://gist.github.com/samueljseay/bd133cbd1cb213b37f7f573351dcccea

EXAMPLE: A BASIC WEBPACK.CONFIG.JS FILE WITH UPDATED INDEX.HTML LAYOUT

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Free Agent Tracker</title>
    </head>
    <body>
        <div id="app"></div>
        <script src="bundle.js"></script>
    </body>
</html>
```

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

module.exports = {
    entry: './src/app.js',    // the entry point of the application
    output: {
        path: path.join(__dirname, 'public'),   // output to public folder
        filename: 'bundle.js'                   // output bundled JavaScript as bundle.js
    }
};
```

IMPORTING 3RD PARTY NPM MODULES INTO WEBPACK

- 3rd party modules are easy to use within your code with Webpack! It is a 2-step process:
  - Install the specific module using **yarn**.
  - Import the module into your script.

- For instance, let's try adding the very popular **validator** module into our code (https://www.npmjs.com/package/validator) for testing and sanitizing input:
  - Step 1: **yarn add validator**
  - Step 2: **import validator from 'validator';**
  - Now you are able to use any of validator's validation and sanitization functionality.

- **Tip:** Remember to install React and ReactDOM, as you will no longer using the CDN with Webpack!
  - **yarn add react react-dom**

ADDING BABEL.JS TO WEBPACK

- Even though we added React and ReactDOM to our project, we still wouldn't have a way to transpile them.
- We can use ***loaders*** in Webpack to transpile our code from specific formats such as JSX.
  - The loader is where we will add Babel.js to our Webpack configuration.
- However, before we do this, we will first need to install some new Babel packages via yarn:
  - **yarn add babel-core babel-loader**
  - These are specific packages to have Babel work with Webpack, meaning we will no longer need to use the Babel CLI command anymore once we update our Webpack config file.

EXAMPLE: WEBPACK CONFIG FILE USING THE BABEL LOADER

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

module.exports = {
    entry: './src/app.js',    // the entry point of the application
    output: {
        path: path.join(__dirname, 'public'),   // output to public folder
        filename: 'bundle.js'                   // output bundled JavaScript as bundle.js
    },
    module: {
        rules: [{
            loader: 'babel-loader',     // the specific loader
            test: /\.js$/,              // the file types to convert
            exclude: /node_modules/     // we don't need to convert anything in this folder!
        }]
    }
};

/*
    NOTE: the above test and exclude property values are what are known as
          REGULAR EXPRESSIONS.
*/
```

ADDING BABEL.JS TO WEBPACK

- You may (or may not) have noticed that when we ran Babel from the command line, we included the type of presets we were using, which in our case were **env** and **react**.
  - We still need to specify these for Babel.js to properly work, and we will declare these within a **.babelrc** file located in the project root.

- This file will contain a single JSON object, containing a ‘presets’ attribute:

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

WEBPACK: SOURCE MAPS

- Source Maps are great to use when we need to track down potential run-time errors inside of our code.
  - Because our React code isn't what is actually being used by the browser, error messages pertaining to the generated bundle.js file aren't really that useful.

- We add a source map to our code by adding the **devtool** property to our Webpack config file. You must choose the specific devtool to use, which in our case will be ***cheap-module-eval-source-map***.
  - Upon addition of this devtool, error messages in the console will now show the specific file that the error originates from instead of the generated code location.

- You can see more information on Source Maps at https://webpack.js.org/configuration/devtool/

EXAMPLE: WEBPACK CONFIG FILE WITH DEVTOOL

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

module.exports = {
    entry: './src/app.js',    // the entry point of the application
    output: {
        path: path.join(__dirname, 'public'),   // output to public folder
        filename: 'bundle.js'                   // output bundled JavaScript as bundle.js
    },
    module: {
        rules: [{
            loader: 'babel-loader',     // the specific loader
            test: /\.js$/,              // the file types to convert
            exclude: /node_modules/     // we don't need to convert anything in this folder!
        }]
    },
    devtool: 'cheap-module-eval-source-map'
};
```

THE WEBPACK DEV SERVER

- Although live-server has been working fine so far, we should move over to using the Webpack dev server, since this is more responsive to code changes.
  - You will need to install the server before you begin via yarn: **yarn add webpack-dev-server**

- Once installed, the dev server can be set up inside of your Webpack config file by setting the **devServer** property.

```
devtool: 'cheap-module-eval-source-map',
devServer: {
    contentBase: path.join(__dirname, 'public')
}
```

- Finally, add your dev server script to your **package.json file:**
  - **“scripts”: {**
    “dev-server”: “webpack-dev-server”
    }

WEBPACK: CONCLUSION

- That's it, we now have Webpack configured to build out a more elegant application!
- We'll continue to add features to Webpack as we move along, specifically with styling, but for now, we have added enough to:
  - Break out our application into separate component files.
  - Convert our JSX to JavaScript via Babel.js
  - Better track down errors by using a Source Map
  - Run our application directly from the Webpack dev server.

THIRD-PARTY COMPONENTS

INTRODUCTION: THIRD PARTY COMPONENTS

- Creating your own components is a key skill to have, but sometimes, you want to use a third-party component so that you do not have to re-create a commonly deployed app feature.
  - In our case, let's look at how we can use a 3rd party to access a modal window.
    - React-modal is a commonly used third-party component for generating and using modal windows. You can see more information on react-modal at https://github.com/reactjs/react-modal

- Before we can use the component, we'll need to install it via **yarn**:
  - **yarn add react-modal**

REACT-MODAL

- If you review the react-modal documentation, you'll see that once you have it installed, you'll first need to import it inside of your script:
  - **import Modal from 'react-modal'**

- Now that you have your modal imported, you'll see that in order to use the modal, it will work with **this.props.children** to render the modal content to the user.

- There will be at least 2 props required for the modal to work:
  - **isOpen**: this prop determines whether or not the modal should be open. Should resolve to a boolean value.
  - **contentLabel**: used for accessibility. Will not show up in the browser.

- In addition to the two required props, react-modal supports some other props:
  - **onRequestClose()**: used if the area outside of the modal is clicked or the escape key is pressed.
  - **onAfterOpen()**: triggered just after the modal is opened.

EXAMPLE: REACT-MODAL

```
<Modal
    isOpen={this.state.modalIsOpen}
    onRequestClose={this.closeModal}
    contentLabel="Show Player Modal"
    className="modal"
>
    <h2>{this.state.curPlayer.name}</h2>
    <h3>{this.state.curPlayer.gender}</h3>
    <p>{this.state.curPlayer.message}</p>
    <button className="button" onClick={this.closeModal}>close</button>
</Modal>
```

STYLING YOUR APP

STYLING OUR APPS WITH REACT

- In order to style our app, we are going to use SCSS to compile into CSS that we will use in our app.
- However, before we can do this, we'll first need to download the specific loaders that we'll need to convert our rules in Webpack. The specific loaders we'll need are:
  - **CSS Loader** - This loader is used to allow Webpack to load our CSS.
    - https://www.npmjs.com/package/css-loader
  - **Style Loader** - This loader actually adds your CSS to the DOM.
    - https://www.npmjs.com/package/style-loader
  - **Sass Loader** - This allows us to load our code into Webpack.
  - **Node-Sass** - This converts the code into regular CSS
  - **yarn add style-loader css-loader sass-loader node-sass**
- Once you have your loaders installed, we can now add them to Webpack as new rules. Since we are applying multiple rules to the same test case (CSS files), we will need to specify more than one rule with **uses** instead of **loader.**

EXAMPLE: WEBPACK.CONFIG.JS WITH NEW SCSS AND CSS RULES

```
module.exports = {
    entry: './src/app.js',    // the entry point of the application
    output: {
        path: path.join(__dirname, 'public'),   // output to public folder
        filename: 'bundle.js'                   // output bundled JavaScript as bundle.js
    },
    module: {
        rules: [{
            loader: 'babel-loader',     // the specific loader
            test: /\.js$/,              // the file types to convert
            exclude: /node_modules/     // we don't need to convert anything in this folder!
        }, {
            test: /\.s?css$/,
            use: [                      // for multiple rules on a test case,
                'style-loader',         // use 'uses' instead of 'loader'
                'css-loader',
                'sass-loader'
            ]
        }]
    },
    devtool: 'cheap-module-eval-source-map',
    devServer: {
        contentBase: path.join(__dirname, 'public')
    }
};
```

WAIT!! BEFORE WE MOVE ON...

LET'S HAVE A QUICK CHAT ABOUT OUR PROJECT STRUCTURE

REACT PROJECT STRUCTURE

- So far, we've held all of our components inside of a single file. As mentioned, this is not a scalable and maintainable solution in the real world.
  - However, now that we have Webpack, we can do something about it! We can now store all of our components into separate files inside of a **components** folder, and import them whenever needed.

- Like with our components, we will be doing the same with our SCSS files by using the built-in **@import** statement in SCSS.
  - For more information on SCSS, please see http://sass-lang.com/

SETTING UP YOUR APP'S STYLE

- To begin, we'll set up a main **styles** folder inside of our **src** folder. Inside of this folder, we'll create a new file called **style.scss**. This will serve as the base file for our styles that references all of our sub-scss files.
- Now that we have our base file set up, it is good practice to separate each individual component's styles into it's own file, as well as have a base style to fall back on. Your initial SCSS file structure should look like:
  - **/styles**
    - **/base**
      - **_base.scss**
    - **/components**
      - **_header.scss**
      - **…the rest of your component styles**
  - **style.scss**
- Inside of your **style.scss** file, import each individual style using **@import**. You do not need to include the .scss file extension:
  - **@import './base/base';**

SETTING UP YOUR APP'S STYLE

- There are many ways to set up your CSS structure, and each developer tends to have a way of their own for tackling this.
- With SCSS, you can take advantage of a bunch of built-in features, such as:
  - Importing partial SCSS scripts (as we've seen)
  - Declaring variables to prevent repetition
  - Nesting rules
- A good starting practice is to assign a unique class name to each component's enclosing tag, and use nested styles to style each element.

EXAMPLE: SETTING UP YOUR APP'S STYLE (SCSS)

```scss
/*      /styles/components/_header.scss     */

.header {

    background: #b3070b;
    margin-bottom: 4rem;
    padding: 1.6rem 0;
    color: #FFFFFF;

    h1 {
        font-size: 3.2rem;
        padding-left: 1rem;
        margin: 0;
    }

    h2 {
        color: #F7F7F7;
        padding-left: 1rem;
        font-size: 1.6rem;
        font-weight: normal;
        margin: 0;
    }
}
```

EXAMPLE: SETTING UP YOUR APP'S STYLE (COMPONENT)

```
import React from 'react';

const Header = (props) => {
    return (
        <div className="header">
            <h1>{props.children}</h1>
            <h2>{props.subtitle}</h2>
        </div>
    );
}
Header.defaultProps = {
    subtitle: "Get active after work!"
};

export default Header;
```

SETTING UP YOUR APP'S STYLE

- Before we dive too deep into setting up our app style, one more thing to note is the ability to reset our styles from the browser defaults.
  - This is typically done by loading in a standard **reset.css** file as the first CSS loaded.

- We'll use **normalize.css** to set our initial CSS values, as it can be easily done from command line.
  - **yarn add normalize.css**

- Once it's added, you can add the normalized CSS to your file by importing the following line of code into your **app.js**:
  - **import 'normalize.css/normalize.css';**

RESOURCES

- Webpack:
  - Official Page: https://webpack.js.org/
  - webpack.config.js Options: https://webpack.js.org/concepts/
  - Source Maps: https://webpack.js.org/configuration/devtool/
- ES6:
  - Import: https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/import
  - Export: https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export
  - CheatSheet: https://gist.github.com/samueljseay/bd133cbd1cb213b37f7f573351dcccea
- NPM Validator Module: https://www.npmjs.com/package/validator
- React Modal: https://github.com/reactjs/react-modal
- React Styles:
  - CSS Loader: https://www.npmjs.com/package/css-loader
  - Style Loader: https://www.npmjs.com/package/style-loader
  - Sass Loader: https://www.npmjs.com/package/sass-loader
- Sass: http://sass-lang.com/

DISCLAIMER

- Copyright 2018 Collegio Technologies Inc.
- All content within this presentation has been compiled by Collegio Technologies Inc. All material has been sourced through open source resources and source code and as such conforms to Canadian copyright laws and regulations. All rights reserved.