Node + React

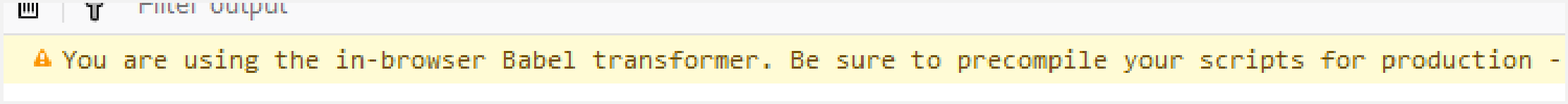
IMY 220 ● Lecture 19

Recap

We have already looked at what Nodejs is and how to set up a simple webserver using NPM to install some modules

We've also looked at how React works, but had to put everything in a ***babel*** script tag which, as mentioned, is not recommended for production code

```
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
<script type="text/babel">
    // All your JS goes here
</script>
```



Node + React

Today we will go through a simple example that uses Node to server a page that uses React and JSX

To do this, we will need two extra tools:

- Babel

- Webpack

Babel

“Babel is a tool that helps you write code in the latest version of JavaScript. When your supported environments don't support certain features natively, Babel will help you compile those features down to a supported version.”

https://github.com/babel/babel

However, Babel also contains presets that allow you to use other syntaxes as well, such as JSX

Babel can thus be used to compile JSX code into valid JS code

Babel

To start off, we're just going to install and use Babel to run some ES6 code in Node

We're also going to use Express to create the webserver, so start by using npm init to initialise a *package.json* file and installing Express using npm install express

Create a new file called *server.js* and add the following code (from L18 - NPM)

```
const express = require("express");

//CREATE APP
const app = express();

//SERVE A STATIC PAGE IN THE PUBLIC DIRECTORY
app.use(express.static("public"));

//PORT TO LISTEN TO
app.listen(1337, () => {
      console.log("Listening on localhost:1337");
});
```

Babel

Also create a new directory for the static content, in this case a folder named *public*, and create a new HTML file inside called *index.html*

```
<!DOCTYPE html>
<html>
<head>
        <meta charset="utf-8" />
        <title>Node + react</title>
</head>
<body>
        <h1>Our first React app in Node</h1>
        <div id="root"></div>
        <!-- We'll get to root when we start adding React elements -->
</body>
</html>
```

Babel

To check if our Babel works, we're going to add some ES6 to our server setup, so replace…

```
const express = require("express");
```

with…

```
import express from "express";
```

If we run our server now, it won't work, since Node's most recent stable version doesn't support import syntax (yet)

Babel

To get it to work, we need to install and set up three Babel tools: babel core, the CLI (command line interface), and the environments preset

npm install --save-dev @babel/cli @babel/core @babel/preset-env

(--save-dev is used when a tool is used for development purposes, as opposed to being required for the operation of the web application itself, for example, Express)

The @ means that the packages are **scoped**, which is a way to group related NPM packages together

https://docs.npmjs.com/misc/scope

Babel

Babel also requires a configuration file called *.babelrc* to work, so create a new file called *.babelrc* and add the following code

```
{
  "presets": ["@babel/preset-env"]
}
```

This tells Babel to transpile all ES6 code by default using its environment preset (Babel has many other presets, including the react-preset)

HINT: to create this file on Windows, open Command Prompt and type copy nul .babelrc

Babel

The last thing we need to do is tell Babel to transpile the code

To do this, we can specify a script to run in our *package.json* file

```
"scripts": {
      "build": "babel server.js -d dist"
}
```

If we run this script with npm run build, Babel transpiles our *server.js* and creates a new syntactically valid *server.js* inside a new directory called *dist*

Babel

We can actually create one script to run each time we want to test our project

```
"scripts": {
    "build": "babel server.js -d dist",
    "start": "npm run build && node dist/server.js"
},
```

Now, calling npm start first calls build, which transpiles *server.js* and then calls runs the transpiled *server.js* inside the *dist* directory with node

Node + React

So far we have a server that servers a static HTML file when we visit **localhost:1337**

We want it to also include some markup we define using React

Let’s use a simple React component for now. Create a new directory called *src* and add a file called *index.js* which defines and renders a simple React component (next slide)

```
import React from "react";
import ReactDOM from "react-dom/client";

class Greeting extends React.Component{
      render(){
            return (
                  <div>
                        <h2> Hello React! </h2>
                  </div>
            );
      }
}

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<Greeting />);
```

Include the React packages (which we'll install next)

This refers to the "div#root" in our *index.html*

Node + React

To be able to use React and JSX, we first have to install React's packages

(Recall from *L19 – React* that React's core functionality is split into two packages called React and ReactDOM)

npm install react react-dom

Node + React

We also have to install a few more devDependencies, namely Babel's react-preset, Babel's loader package for Webpack, and Webpack itself

"webpack is a module bundler. Its main purpose is to bundle JavaScript files for usage in a browser, yet it is also capable of transforming, bundling, or packaging just about any resource or asset. "

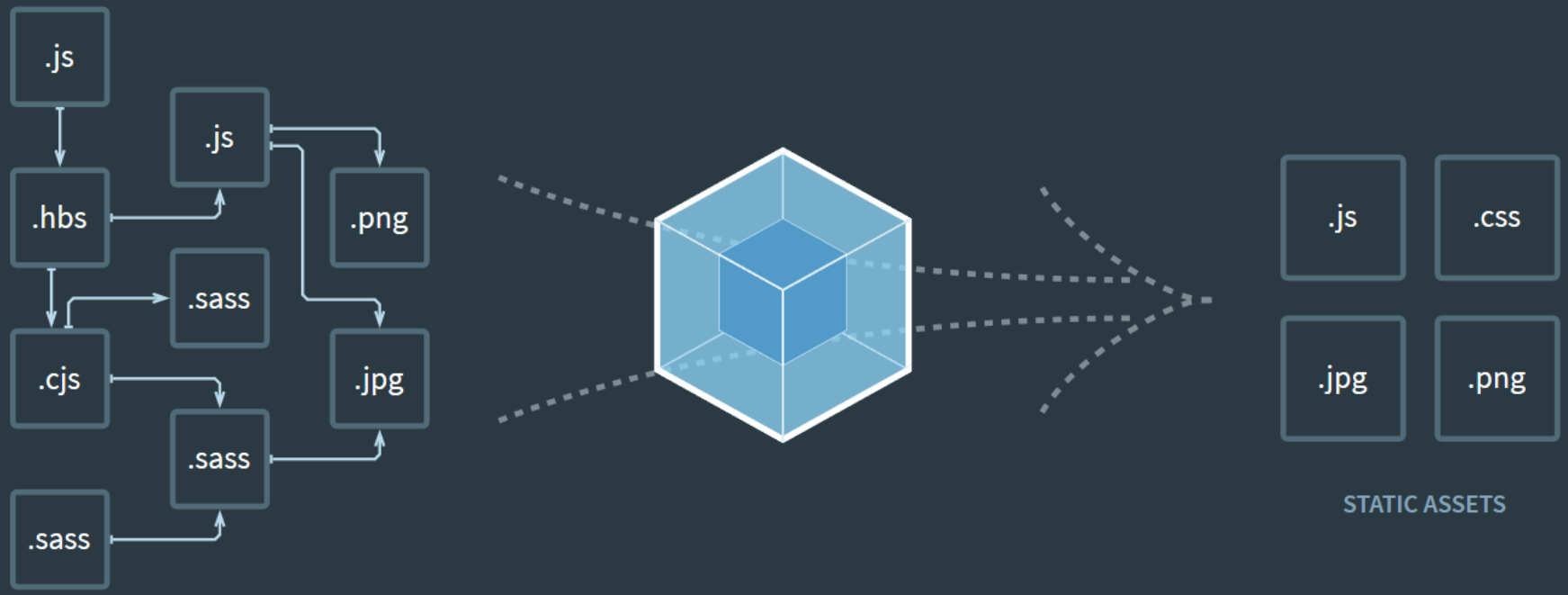
https://github.com/webpack/webpack

npm install --save-dev @babel/preset-react babel-loader webpack webpack-cli

webpack

DOCUMENTATION CONTRIBUTE VOTE BLOG

bundle your scripts



Webpack

Webpack takes one/more files to be bundled (in our case just *index.js* so far) and optionally uses some plugins, loaders, transpilers, etc. to deliver an output file that is usable by our main webpage files, in this case by *index.html*

Similar to Babel's *.babelrc* file, Webpack requires a configuration file to work properly

Create a new file called *webpack.config.js* and add the following…

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


module.exports = {
        entry: "./src/index.js",
        output: {
                path: path.resolve("public"),
                filename: "bundle.js"
        },
        mode: "development",
        module: {
                rules: [
                        {
                                test: /\.js$/,
                                exclude: /node_modules/,
                                use: {
                                        loader: "babel-loader"
                                }
                        }
                ]
        }
}
```

Webpack

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

NPM package that provides utilities for working with directory paths

We haven’t actually installed this one yet, so first we need to do that

npm install path

Webpack

```
module.exports = {
      ...
}
```

Webpack expects an exported JS object that defines all the parameters for bundling

Webpack

```
entry: "./src/index.js",
```

The entry defines where we want to start our application bundling process

Since our *index.js* file is going to render all the JSX, we provide a path to that file

Webpack

```
output: {
          path: path.resolve("public"),
          filename: "bundle.js"
     },
```

Here we are telling Webpack that we want the resulting bundle to be exported in the *public* directory as a new file named *bundle.js*

Webpack

```
mode: "development",
```

This tells Webpack what we are using the bundler for, which it can then use to adjust its optimisation process

(This isn’t required, but Webpack gives you a warning if you leave it out)

Webpack

```
module: {
            rules: [
                    {
                            ...
                    }
            ]
        }
```

This is where we define the behaviour that we want from Webpack in the form of the rules it has to follow for transformations

Webpack

```
test: /\.js$/,
exclude: /node_modules/,
```

test tells Webpack to only bundle *.js* files

(We can change the regex to specify other file extensions, such as *.jsx*)

exclude tells Webpack not to bundle the files inside the *node_modules* directory (which would have made the bundling process very slow)

Webpack

```
use: {
      loader: "babel-loader"
}
```

Finally, use tells Webpack what plugin, loader, etc. to use for the files we specified

In this case, we are telling it to use babel-loader to transpile all *.js* files

Webpack

We also need to tell Babel that we want to transpile react using the Babel react-preset

Inside *.babelrc*, add the react-preset as a preset to use during transpiling:

```
{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

Webpack

Finally, since we know our bundled JS is going to be in a file called *bundle.js* in the *public* directory, we can include that file inside index.html (which is also inside *public*)

```
<body>
      <h1>Our first React app in Node</h1>
      <div id="root"></div>

      <script type="text/javascript" src="bundle.js"></script>
</body>
```

Webpack

To create *bundle.js* we need to run Webpack. Specifically, the file we want to run is *node_modules/.bin/webpack*.

So, we could run the ./node_modules/.bin/webpack command every time we wanted to bundle our code (in other words, every time we changed our React files)

Luckily NPM has a built-in command for executing binaries from packages, called npx. We can thus use npx webpack to bundle our code.

Webpack

After successfully doing this, there should be a file in *public* called *bundle.js*

Thus, we should now be able to run our server again and see the resulting JSX component in the webpage we're serving

Webpack

Same as with our first Babel example, we could add the bundling script to our *package.json* file to combine all the transpiling and running of the server into a single command line command

```
"scripts": {
    "build": "webpack && babel server.js -d dist",
    "start": "npm run build && node dist/server.js"
},
```

This way, every time we run npm start, all the build code (using Webpack and Babel) creates the transpiled files we need and the server uses those

Webpack

As our files get more and larger, running the webpack command inside our build script may become impractically slow

Webpack has a useful piece of functionality that operates similar to nodemon, in other words, it watches the files to be bundled and automatically re-bundles them when they change

To use this, we can call npx webpack –w in the command line

Webpack

Adding the -w (watch) flag to the script inside *package.json* introduces a problem: it pauses the command line execution at webpack –w

```
"build": "webpack -w && babel server.js -d dist",
"start": "npm run build && node dist/server.js"
```

Execution is paused here

Thus, the script will never get to the part that executes node and serves the webpage

If you want to use webpack -w it is therefore recommended that you open a separate command line window and execute it there (and remove the webpack -w script from *package.json*)

Node + React

The example we have so far is quite simple, what if we wanted to create more complex UI components like we did with the PersonList example from *L20 – React 2*?

If we copy-paste the code from that example into *index.js* (and remember to import React and ReactDOM) it still works

However, this makes the *index.js* file quite bulky, since we are defining three components (in the form of ES6 classes) and rendering a component in the same file

```
class Person extends React.Component {
    render(){
        return (
            <li>{`${this.props.person.name[0]}. ${this.props.person.surname}`}</li>
        );
    }
}

class AddPersonForm extends React.Component{
    constructor(props){
        super(props);
        this.submit = this.submit.bind(this);
        this.nameInput = React.createRef();
        this.surnameInput = React.createRef();
    }

    submit(e){
        e.preventDefault();
        let name = this.nameInput.current.value;
        let surname = this.surnameInput.current.value;
        this.props.onNewPerson(name, surname);
    }

    render(){
        return (
            <form onSubmit={this.submit}>
                <input type="text" ref={this.nameInput} /> <br/>
                <input type="text" ref={this.surnameInput} /> <br/>
                <input type="submit" value="Add" />
            </form>
        );
    }
}
```

```
class PersonList extends React.Component {
    constructor(props){
        super(props);
        this.state = {people: this.props.people || []};
        this.addPerson = this.addPerson.bind(this);
    }

    addPerson(name, surname){
        this.setState({people: [...this.state.people, {name, surname}]});
    }

    render(){
        return (
            <div className="container">
                <h1>
                    {this.state.people.length} in the list:
                </h1>
                <ul>
                    {this.state.people.map( (person, i) => <Person key={i} person={person} /> )}
                </ul>
                <div>
                    <AddPersonForm onNewPerson={this.addPerson} />
                </div>
            </div>
        );
    }
}

var peopleList1 = {
    name: "Troy", surname: "Barnes",
    name: "Abed", surname: "Nadir"
}

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<PersonList />);
```

Node + React

A better way to do this would be to create a separate file for each component and to import the different components as each file needs it

This way, each component is logically separated into its own file and *index.js* simply renders the main component

Start by creating a new folder for the component declarations (we'll call it *components*)

Node + React

Move each class declaration into its new file. We'll call each file by the name of the class that it's declaring as a JS file (e.g. *Person.js*)

Since we are declaring a component as a class that extends React.Component we still need to include React, but if we only want to define a component and not render it, we don't need to include ReactDOM

We also want to export the class declaration for the component. We could do this using default exports, but since this is somewhat of a bad practice, we'll export the class definition instead.

Node + React

Side note: there is some debate regarding the use of different file extensions for defining components, specifically the use of *.js* and *.jsx* files

https://github.com/airbnb/javascript/pull/985

It is worth noting at this point that the extension doesn't affect functionality, since everything is transpiled into *bundle.js* anyway

Node + React

```
// Contents of Person.js inside the new "components" directory

import React from "react";
// Since we're only defining a class but not rendering it, we don't need
// to import ReactDOM here

export class Person extends React.Component {
   render(){
      return (
         <li>{`${this.props.person.name[0]}. ${this.props.person.surname}`}</li>
      );
   }
}
```

Node + React

```
// Contents of AddPersonForm.js inside the new "components" directory


import React from "react";


export class AddPersonForm extends React.Component{
   // rest of class definition goes here (see L20 - React 2)
}
```

Node + React

Since PersonList in our example makes use of our other components (Person and AddPersonForm), we need to import them

Recall from *L11 ES6-2* that we need to destructure individually exported types such as class definitions when we import them

Node + React

```
// Contents of AddPersonForm.js inside the new "components" directory

import React from "react";

import {Person} from "./Person";
import {AddPersonForm} from "./AddPersonForm";

export class PersonList extends React.Component {
   // rest of class definition goes here (see L20 - React 2)
}
```

Node + React

And finally, since PersonList imports the other two components, we only need to import PersonList when we want to render it inside *index.js*

```
import React from "react";
import ReactDOM from "react-dom/client";


import {PersonList} from "./components/PersonList";


const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<PersonList />);
```

Node + React

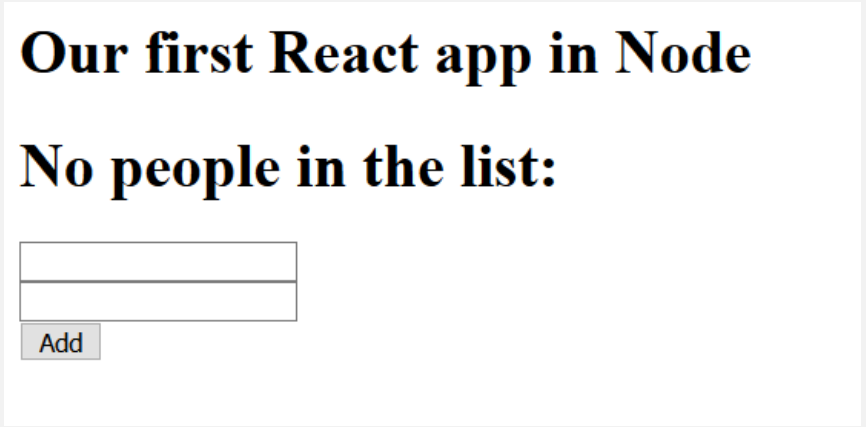
Since we're bundling 4 files now (as opposed to just *index.js* before), does that mean that we have to add them to our *webpack.config.js*?

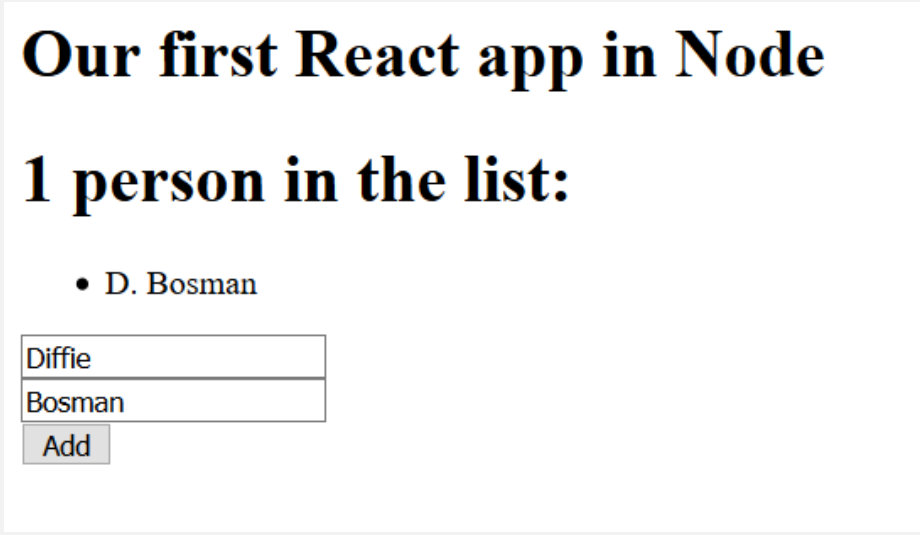
Luckily no, since we defined *index.js* as our entry point for bundling and *index.js* imports *PersonList.js*, which in turn imports *AddPersonForm.js* and *Person.js*

Thus our *webpack.config.js* remains unchanged

Node + React

You should be able to re-bundle and restart the server to see the functionality from our example





Prop Types

Since JS is a loosely typed language, you can supply a variable with a value of any type without getting an error

This can lead to some confusion, such as if a variable is given a value which is syntactically acceptable, but leads to logic errors

(For example, unknowingly adding a numerical string to an integer)

To prevent this type of confusion, React provides functionality for Property Validation in the form of Prop Types.

Prop Types

Prop Types used to be part of React by default, but has since moved into its own prop-types library

npm install prop-types

We also need to import it into any files that are going to validate props

```
import PropTypes from 'prop-types';
```

Prop Types

We can use PropTypes to specify what type a specific prop should be

For example, we know that in our Person class we're expecting this.props.person to be an object

Thus we can specify that the prop must only accept values that are JS objects by declaring a class static property for the component

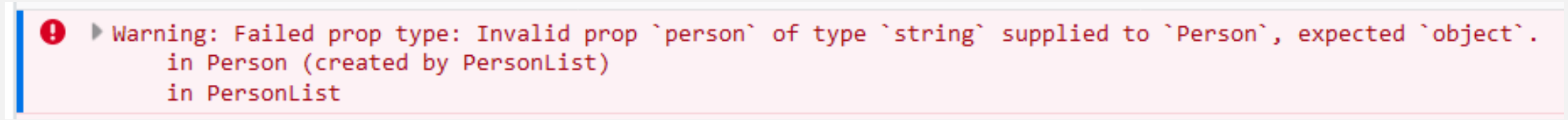
```
import React from "react";
import PropTypes from "prop-types";

export class Person extends React.Component {
      render(){
            return (
                  <li>{`${this.props.person.name[0]}.
                  ${this.props.person.surname}`}</li>
            );
      }
}

Person.propTypes = {
      person: PropTypes.object
}
```

Prop Types

If we now try to create a Person component with a person prop that is anything except a JS object, we'll get a descriptive error

For example, if we try to add one as follows inside *PersonList.js*

```
{this.state.people.map( (person, i) => <Person key={i} person="Name" /> )}
```



Prop Types

It is always a good idea to do this, as it can help debug otherwise tricky errors

The Prop Types library has many built-in type checks, such as:

- PropTypes.array
- PropTypes.bool
- PropTypes.func
- PropTypes.number
- PropTypes.object
- PropTypes.string
- PropTypes.symbol

Prop Types

It is also a good idea to validate the existence of a required prop using PropTypes.isRequired

You can also chain Prop Type validation and add validation for multiple props simultaneously, for example:

```
Person.propTypes = {
        person: PropTypes.object.isRequired,
        example2: PropTypes.func.isRequired
}
```

You can find many more examples of Prop Type validation here:

https://react.dev/reference/react/Component#static-proptypes

That’s it!

References

Banks, A. & Porcello, E. 2017. *Learning React: Functional Web Development with React and Redux*. O'Reilly Media, Inc.

https://babeljs.io/docs/

https://webpack.js.org/concepts/

https://github.com/webpack/webpack

https://github.com/babel/babel-loader

https://www.npmjs.com/package/npx

https://react.dev/reference/react