**REACTJS**

**PART 1**

**Setup**

1. Create project folder
2. Npm init //inside project folder
3. Npm install react react-dom //for production
4. Npm install webpack webpack-cli webpack-dev-server babel-loader babel-preset-env babel-preset-react babel-preset-stage-2 -D
5. Structure your code:

Src > APP > JS > index.js

> CSS > style.css

Index.html

Webpack.config.js

README.MD

Webpack.config.js

Var webpack = require(‘webpack’);

Var path = require(‘path’);

Var DIST\_DIR = path.resolve(\_\_dirname, “dist”);

Var SRC\_DIR = path.resolve(\_\_dirname, “src”);

Var config = {

Entry: SRC\_DIR + “/app/index.js”,

Output: {

Path: DIST\_DIR + “/app”;

Filename: “bundle.js”,

publicPath: “/app/”

};

Module: {

Rules: [

{

Test: /\.js?/,

Include: SRC\_DIR,

Loader: “babel-loader”,

Query: {

Presets: [“react”, “env”, “stage-2“]

}

}

]

}

};

Module.exports = config;

//make index.js file => create – console.log(‘worked?’);

//in index.html – make script tag at end of body with src=”/app/bundle.js”

Package.json

//remove test under scripts. And include the following under scripts:

“start”: “npm run build”,

“build”: “npx webpack -d && cp src/index.html dist/index.html && webpack-dev-server –content-base src/ --inline”,

“build:prod”: “npx webpack -d && cp src/index.html dist/index.html”

* In terminal: npm start

**Simple set-up**

Npm install -g create-react-app //only do once to install on machine globally

Create-react-app app-name

Cd app-name

Npm start

* Look at README.MD for instructions
* Work with app.js and index.js

**REACTJS – TUTORIAL**

**Overview**

ReactJS is a Javascript library used to create user-interfaces. It was created and is maintained by Facebook. React can be run on the client and the server.

React has something called the virtual DOM, which only updates the parts that need to be re-rendered, which makes it a lot faster than the standard JS DOM.

Why use ReactJS

* Simple – React manages all of the UI updates for you
* Declarative – React knows to refresh only the changed parts of the app
* Readable and re-usable – Everything is considered a “component” and is self-contained, making it easy to read & to re-use code

JSX

React use Javascript Syntax Extension – which is a tag syntax very similar to XML & HTML.

The babel compiler transforms JSX code in to JS code. The babel compiler also allows you to write ES6 syntax (transforms to earlier ES versions, as some ES6 syntax is not yet supported by all browsers). The babel-compiler also works very well with webpack.

**ReactJS – Documentation/official tutorials**

**Example**

ReactDOM.render(

<h1>Hello World!</h1>,

Document.getElementById(‘root’)

);

//where *root* is the element in the html file we wish to render to.

**Introducing JSX**

Const element = <h1>Hello, World!</h1>; //this var stores JSX syntax

You can embed any JS expression in JSX by wrapping it in curly braces, for example:

Function formatName(user) {

Return user.firstName + ‘ ‘ + user.lastName;

}

Const user = {

firstName: “Ali”,

lastName: “Issaee”

}

Const element = (

<h1>

Hello, {formatName(user)}!

</h1>

);

ReactDOM.render(

Element,

Document.getElementById(‘root’)

);

//const element is over multiple lines for readability

JSX is an expression too, this means you can use JSX inside of *if statements* and *for loops*, assign it to *variables*, accept it as *arguments*, and *return* it from functions:

Function getGreeting(user) {

If(user) {

Return <h1>Hello, {formatName(user)}!</h1>;

}

Return <h1>Hello, stranger!</h1>;

}

You can specify attributes with JSX. Since JSX is closer to JS than XML & HTML, ReactDOM uses camelCase property naming convention.

You may use quotes to specify string literals as attributes:

Const element = <div tabIndex = “0”></div>

You may also use curly braces to embed JS expressions into attributes:

Const element = <img src={user.avatarUrl}></img>;

*\*Don’t put quotes around curly braces! \**

JSX represents objects. Babel compiles JSX down to React.createElement() calls. The following 2 examples are identical:

const element = (

<h1 className = “greeting”>

Hello, world!

</h1>

);

Const element = React.createElement(

‘h1’,

{className: “greeting”},

‘Hello, world!’

);

**Rendering elements**

Elements are the smallest building blocks of react apps, an element described what you want to see on the screen:

Const element = <h1>Hello, world</h1>;

Elements are what components are “made of”.

To render a react element into a root DOM node, pass both to ReactDOM.render():

Const element = <h1>Hello, World!</h1>;

ReactDOM.render(element, document.getElementById(‘root’));

Updating the rendered element

React elements are immutable. Once you create an element, you can’t change its children or attributes. An element is like a single frame in a movie; it represents the UI at a certain point in time.

ReactDOM only updates what is necessary. It compares the element and its children to the previous one and only applies the DOM updates necessary to bring the DOM to the desired state. Inspect the following code to see:

Function tick() {

Const element = (

<div>

<h1>Hello, World!</h1>

<h2>It is {new Date().toLocaleTimeString()}.</h2>

</div>

);

ReactDOM.render(element, document.getElementById(‘root’));

}

setInterval(tick, 1000);

**Components and props**

Components let you split the UI into dependent, reusable pieces, and think about each piece in isolation.

Conceptually, components are like functions. They accept arbitrary inputs (called “props”), and return React elements describing what should appear on screen.

2 ways to define a component; function and ES6 class:

1 – function

Function Welcome(props) {

Return <h1>Hello, {props.name}. </h1>

}

2 – ES6 class

Class Welcome extends React.Component {

Render() {

Return <h1>Hello, {this.props.name}. </h1>;

}

}

//classes have some additional features, discussed in part 2. This booklet continues using functions for readability and learning purposes.

Rendering a component

Previously, we only encountered react elements that represent DOM tags (e..g <div>). However, elements can also represent user-defined components:

Const element = <Welcome name=”Ali” />

//where name is a prop

\*Always start components with a capital letter\*

Components can refer to other components in their output:

Function Welcome(props) {

Return <h1>Hello, {props.name}.</h1>;

}

Function App() {

Return (

<div>

<Welcome name=”Ali” />

<Welcome name=”Lauren”/>

</div>

);

}

ReactDOM.render(

<App />

Document.getElementById(‘root’)

)

Typically, new React apps have a single app component at the very top.

Extracting components

Don’t be afraid to split components in to smaller components.

*Example:*

Function Comment(props) {

Return (

<div>

<div>

<img className = “Avatar”

Src={props.author.avatarUrl}

Alt = {props.author.name}

/>

<div>

{props.author.name}

</div>

</div>

<div>

{props.test}

</div>

<div>

{formatDate(props.date)}

</div>

</div>

);

}

//this component accepts author (which is an object), text (a string), and a date (a date) as props. And describes a comment on a social media website.

//in its current state, it is hard to reuse individual parts of it.

Extracting components:

//extract Avatar

Function Avatar(props) {

Return (

<img className = “Avatar”

Src={props.user.avatarUrl}

Alt = {props.user.name}

/>

);

}

//to simplify Comment, replace <img /> with:

<Avatar user={props.author} />

//next we will extract a UserInfo component that renders an Avatar next to the user’s name:

Function UserInfo(props) {

Return (

<div>

<Avatar user={props.user} />

<div>

{props.user.name}

</div>

</div>

);

}

//this lets us simplify Comment even further, down to:

Function Comment(props) {

Return (

<div>

<UserInfo user={props.author} />

<div>

{props.text}

</div>

<div>

{formatDate(props.date)}

</div>

</div>

);

}

Extracting components might seem like grunt work at first, but having a palette of reusable components pays off in larger apps. A good rule of thumb is if a part of your UI is used several times (Button, Panel, Avatar etc…), or is complex enough on its own (App, FeedStory, Comment), it is a good candidate to be a reusable component.

\*Props are read-only, when you declare a component as a function or a class, it must never modifty its own props\*