# MODERN REACT

## REACT CODEPEN

- Need to go to settings and add the react and reat-dom libraries

## WHAT IS IT

- JS library

- Purpose to show content(HTML) to users and handle user interaction

- React can work by itself, but to make it faster we pull in other libraries, packages, servers, databases, etc.

- Components made using JS class or function

- JSX

- Components can be used to make event handlers

- Need to add React library becayse it contains info on what component is and how to make them work together

- ReactDOM library knows how to make a component and make it show up on the DOM

## CHECK NODE JS VERSION

node –v

## GENERATE REACT PROJECT

### INSTALL CREATE-REACT-APP PACKAGE

sudo npm install –g create-react-app

- npm = ‘Node package manager’

- Installs JS packages

- ‘-g’ – installs globallly so can run from terminal

### CREATE-REACT-APP TERMINAL

create-react-app filename

### WHAT IS GOING ON

- This will install a bunch of react dependencies

#### BABEL

- Not all browsers are up to date with ES2016 and later

- Babel takes any version of JS and spits out newer version

- We write our code 🡪 passed into Babel 🡪 Spits out ES5

- It also turns JSX into normal ES5 Javascript code

<https://babeljs.io/> 🡪 Use ‘try it out’ to see how its converting

### FILES CREATED AFTER INSTALL

#### src

- Usually delete the automated files, but this is where our main files are

#### public

- Where static files stored (html, images)

#### node\_modules

- Each folder represents the different dependency

#### package.json

- Lists out the dependencies and configures the project

#### package-lock.json

- Lists exact versions of dependecies

#### READ-ME

- Info on how to run app

#### .gitignore

- Files git should ignore

### NPM START

- Type this in terminal to start the app on localhost: 3000

### NPM PRETTIER

- Automatically makes code look nice

npm install --global prettier

## JS MODULE SYSTEMS

### IMPORTS

- Every file is its own universe and isnt shared with other files, so we have to import things to every file

**import React from 'react';**

**import ReactDOM from 'react-dom';**

- Import gets code from some other file or dependecy, ‘React’ is the name where we attach everything from the ‘react’ dependecy (assign the library to the same name), ‘from’ means we are about to list a dependency, ‘react’ goes to node\_modules and finds the folder to get all the methods

- Thus we can import any of the dependecies found in the node\_modules folder

- Import references ES2015 Modules, ‘require’ used for CommonJS modules

### GENERAL FILE SETUP

#### index.js

import React from 'react';

import ReactDOM from 'react-dom';

#### COMPONENTS

- Produces HTML to show to user (JSX)

- Event handles

- Create a react component

*const* Greeting = () *=>* <h1>Hello World</h1>;

- Take the react component and show it on the screen

- The index.html file which is made with the create-react-app, we will pass the render to the root within

ReactDOM.render(

<*Greeting* />,

document.getElementById('root')

)

#### RENDERING

- Also when returning just html not in component, best to wrap it in parentheses

**const** App = () **=>** {

return (

<div>

<label class="label" for="name">Enter name:</label>

<input id="name" type="text" />

<button style="background-color: blue; color:white">Submit</button>

</div>

)

}

## COMPONENTS

NESTING - Place them in eachother to make useful application

REUSABILITY - Want to make components that can be easily reused inside application

CONFIGURATION - Need to configure component when it is created

### CREATING COMPONENT

- Identify the JSX that appears to be duplicated

- What is its purpose? Think of descriptive name for it

#### CREATE COMPONENT FILE

- Create it in src file (or components file within)

- Import ‘react’

- Create new file to house this new component – should have same name as component

- Create new component in the new file, paste the JSX into it

#### EXPORT FROM COMPONENT FILE

- Export the Component from the component file to make it available to index.js

export default Component;

#### IMPORT TO MAIN FILE

- Import the Component to link it and make it available

- Known as component nesting

- Don’t need the file extension (.js) since ‘webpack’ automatically does this

import Component from './Component'

#### USE COMPONENT IN MAIN FILE

- Show as JSX component

<Component />

#### COMPONENT HEIRARCHY

- Child components are within parent components

### PROPS (PROPERTIES)

- Make the new component configurable using Reacts ‘props’ system

- System for passing data from parent to the child

- Goal is to customize or configure a child component (how it looks or how users interact)

- So props can pass in a name or date from the parent to the child

#### PROVIDE INFORMATION FROM PARENT TO CHILD

- In instance of the parent component

- Give name of prop

- give value to prop

**<Component name=’value’ />**

- Can reference JS variable as well, use curly braces

**<Component name={variable} />**

##### PASS IN MULTIPLE VALUES

- Make into multiline component

- Only need spac between values

<Component

author="Davis"

timeAgo="Today at 2:45PM"

comment="I dig you"

image={faker.image.avatar()}

/>

#### CONSUME INFORMATION FROM PROP IN CHILD

- In the Component file, pass the ‘props’ variable in as an argument to the function which will then hold the keys and values which were passed in from the parent

**const CommentDetail = props => {**

**return (**

**<a href="/" className="author">**

**{props.author}**

**</a>**

**)**

**}**

### COMPONENT REUSE

- Can pass in props in a second way by using Components like we do html elements

- The Component ‘CommentDetail’ will then be available on the props object (on the ‘children’ property)

<Component>

{*/\* This component will exist on the props object \*/*}

<CommentDetail author="Brendan" timeAgo="Today at 4:45PM" comment="I like you" image={faker.image.avatar()}/>

</Component >

- Thus to use in the child, the object will be on the props.children in the Component file

const Component = () => {

return (

)

}

<div className="content">{props.children}</div>

### CLASS VS. FUNCTIONAL COMPONENTS

- Functional components when show simple things

- Class components good for everything else (complex logic, respond to user input, etc.)

- Get easier time organizing code

- Can use ‘state’ which is easier to handle user input and update app

- Understand lifecycle events (easier to do tasks when first starts)

### CLASS COMPONENT

- Use to manage feedback from the user and use event handlers to change things

- Borrow methods from React.Component (superclass)

#### RULES

- Must be JS class

- Must extend subclass (React.Component)

- Must define ‘render’ method that returns some amount of JSX

## STATE

- Only usable with class components (can use with functional components with hook system)

- You may confuse props with state (but they are different)

- State is a JS object which contains data relative to a component

- Updating state on a component will cause the component to almost instantly rerender

- State must be initialized when component is first created

**- State can only be updated using function ‘setState’**

### INITIALIZATION

#### INITIALIZING STATE THROUGH CONSTRUCTORS

- Initialize when component first created

- Create state object within the constructor of the component

- The constructor is specially named and belongs to JS, called everytime new instance of the component called

- One way to initialize state

- Need to call ‘super(props) in the constructor to call the parents constructor (React.Component)

**class App extends React.Component {**

**constructor(props){**

**super(props);**

**this.state = {**

**property: null**

**}**

**}**

**render() {**

**}**

**}**

#### ALTERNATE STATE INITIALIZATION

- Placing the following outside the constructor will be the same as setting state within the constructor

- Babel will still build the constructor with super(props) and set state, so this makes code smaller

**- THUS DON’T NEED THE CONSTRUCTOR**

**state = { lat: null, errorMessage: ''}**

### UPDATING STATE PROPETIES

- Can reference any property of state in the component (in any function)

- ‘state’ property is from React.Component

this.state.property

- Any time we update the state, the component almost instantly rerenders

- Don’t want to initialize anything in the render() method since it is called all the time

- Use ‘setState()’ method which is from React.Component

- Don’t directly assign to the state (only in the constructor, so ie don’t do this.state = value)

this.setState({ property: value });

- This will instantly cause component to rerender

- When setState, only edit the properties which are within the curly brackets, never deleting or changing ones that arent referenced

### HANDLING ERRORS

- Want to rerender the component and tell the user something went wrong

- Do this by updating state

### LIFECYCLE METHODS

- Function can optionally define in class components

- Called during component lifecycle

🡪 Constructor called 🡪 render method called

- If we define function in the component, it will be called when the component is created

- componentDidMount()

- Can define function which is called when component is updated (define in component)

- render() called first

- componentDidUpdate()

- Can define method which can unmount a component and not be called anymore

- componentDidUnmount()

#### WHY USE LIFECYCLE METHODS

- shouldComponentUpdate, getDerivedStateFromProps, getSnapshotBeforeUpdate methods rarelly used

##### COMPONENTDIDMOUNT

- Recommended to do data loading (loading geolocation, etc.) in here instead of constructor

- Better to centralize here to have more clear code

- Thus the constructors only job now is to setState

##### componentDidUpdate

- Data loading when state/props change (like user input)

##### componentWillUnmount

- Good place to do clean up

- Not used as much anymore

### PASSING STATE AS PROPS

- Can pass properties from state into child components in render()

**return <SeasonDisplay lat={this.state.lat}/>**

- In the child component, the props argument will now contain this property

### SPECIFY DEFAULT PROPS

- Just incase don’t put a prop in the component, but if input one then the default is overrided

- Can specify in an or statement in the Component file

{props.message || ‘Loading…’}

#### defaultProps

- Can add the property to this object

Component.defaultProps = {

message: ‘loading…’

};

### AVOID CONDITIONALS IN RENDER

- Create helper function where the conditions are placed in so can easily add className to one div where the helper method is placed in indtead of doing individually

## STYLING

### IMPORT CSS FILE

- Can import into the Component which uses the css

- Webpack sees we are importing css file, takes it and places in index.html file

import './SeasonDisplay.css';

#### NAMING

- Have the root elements class name the same as the component

#### REACT INLINE STYLES WITH STYLE PROPERTY

- In JSX style expects JS object and not string since the way jsx is transpiled

- The first curly brace shows JS variable is going to be referenced, the second brace represents JS object which is what were using for styling

- Convention to use double quotes for a string (class, id, type), then for any non-JSX value use single quotes (styling)… not mandatory

<h1 style={{color:’FF800’, backgroundColor:’yellow’}}>

- Realize any object property name which contains dash(-) will be taken away and the letter after made capital

background-color 🡪 backgroundColor

- Best to make a styles variable with object then places this in the style property of JSX

var styles = {background:’yellow’}

<h1 style = {styles}>

- Default number value is ‘px’, so if leave out, then it doesn’t need to be a string but if have px then needs to be a string to be a valid object

fontSize: 200 🡪 fontSize:’200px’

- If you want to use em then need to write beside the number in quotes

fontSize:’200em’

### DEFINE HTML CLASS

- The naming convention for all HTML attributes and event references in JSX become camelCase.

- This is because class is a reserved word in JavaScript (we don’t want react to think we are describing new JS class) 🡪 but technology is up to date enough to know the difference now.

🡪 class : className

🡪 onclick : onClick

- Other than that it is pretty similar, attach stylesheet with associated classes and style

### USING JS VARIABLES IN HTML

- Just place any JS in curly braces {} and it will be interpreted

- Any time outside the brackets is JSX land

- Thus any variables from JSX can be used in JS by using them within the curly braces

- Can do this with JS methods as well

#### VALUES JSX CANT SHOW

- Cant show JS object and reference in React text (will get error)

buttonText = { text:’click’}

### FORBIDDEN PROPERTY NAMES

- Look at console in browser and it will show you errors

Warning: Invalid DOM property `for`. Did you mean `htmlFor`?

- So above it wants the html ‘for’ style to be ‘htmlFor’ but it isnt required, but it wants you to change

## REFACTORING

- Use configuration object instead of ternary to set values

**const** seasonConfig = {

*// Objects tell us what text to use and icon*

*// Makes it easier to do any updates*

summer: {

text: 'Lets hit the beach',

iconName: 'sun'

},

winter: {

text: 'Burr, it is chill',

iconName: 'snowflake'

}

}

### PLACING

- Put config files and functional components at the top

- Place the component method at the bottom

## SEMANTIC-UI

<https://semantic-ui.com/>

### CDN

- Place in head tag in

<link rel=”stylesheet” href=”<https://cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.4.1/semantic.min.css>

”>

- Make sure it installed

## FAKER JS

<https://github.com/marak/Faker.js/>

- Help you generate a ton of data to use

- Useful for initial development

- See API Methods section for what you can create

### USE

- Save it to package.json file where it shows we installed

npm install –save faker

import faker from ‘faker’

#### FILE

- Then in say react file

<img alt="avatar" src={faker.image.avatar}/>