**I. Getting Started**

**1. Introduction**

**2. What is React?**

- A Javascript Library for building User Interfaces

- Maintainable, manageable, reusetable

**3. React Web App**

**-** Udemy.com, Reactjs.org

**4. First React Code**

**5. Why Should We Choose React?**

**-** Handle UI state easier

- Focus on Business Logic

- Huge ecosystem

**6. React Alternatives**

**7. Single Page Application and Multi Page Application**

- SPA: only 1 HTML page, content is (re)rendered on Client

- MPA: multi HTML page, content is rerendered on Service

**8. Course Outline**

Getting Started → The basics → Debugging → Styling Components → Components Deep Dive → HTTP Requests → Routing → Form & Validation → Redux → Authentication → Testing Introduction → Deployment → Bonus (Animations, Next Steps, Webpack)

**9. How to get the Most out of This Course**

**II. Refreshing Next Generation Javascript (Optional)**

**1. Module Introduction**

**2. Understanding let and const**

**3. Arrow Function**

**4. Exports & Imports (Modules)**

**-** Default export

import person from './person.js';

import prs from './persons.js';

**-** Named export

import {smth} from './utility.js';

import {smth as Smth} from './utility.js';

import \* as bundled from './utility.js';

**5. Classes**

Note: If a class is extended by another class, the extended class’s constructor must add super() method.

class B extends A {

constructor() {

super();

}

}

**6. Classes, Properties and Methods**

**7. The Spread & Rest Operator**

**-** Operator: …

- Spread: Used to split up array elements or object properties.

const newArray = [...oldArray, 1, 2];

const newObject = {...oldObject, newProp: 5};

**-** Rest: Used to merge a list of function arguments into an array.

function sortArgs(...args) {

return args.sort();

}

**8. Destructuring**

**-** Easily extract array elements or object properties and store them in variables.

- Array Destructuring

[a, b] = ['Hello', 'Hi'];

console.log(a); //Hello

console.log(b); //Hi

- Object Destructuring

{name} = {name: 'Test', age: 28};

console.log(name) // Test

console.log(age) // undefined

**9. Reference and Primitive Type Refresher**

**-** Primitive types: numbers, strings, booleans called primitive types, whenever a variable is reassigned or stored, it will copy the value.

- Reference types: arrays, objects call reference types, whenever a variable is reassigned or stored, it will copy the pointer that points to the object in memory.

**10. Refreshing array functions**

**-** Functions use with array

const numbers = [1, 2, 3];

const doubleNumArray = numbers.map((num) => {

return num \* 2;

});

**10. Wrap up**

**III. Base Feature and Syntax**

**1. Module Introduction**

**2. The Build Workflow**

**-** Why

+ Optimize code

+ Use Next-Gen Javascript Features

+ Be More Productive

- How

+ Use Dependency Management Tool: npm or yarn

+ Use Bundler: Webpack

+ Use Compiler (Next-Gen Javascript): Babel + Presets

+ Use a Development Server

**3. Create React App**

**-** Installing **create-react-app**

npm install -g create-react-app

- Create application:

create-react-app <project-name>

**4. Folder Structure**

**5. Understanding Component Basics**

**-** Root component which should nested all the other components

ReactDOM.render(<React.StrictMode><App /></React.StrictMode>, document.getElementById('root'));

- Defining a React component

+ Functional component

function App() {

return (

<div className="App">

<h1>This is a React App</h1>

</div>

);

}

+ Class-based component

class App extends Component

{

render() {

return (

<div className="App">

<h1>This is a React App</h1>

</div>

)

}

}

**6. Understand JSX**

- **render()** function in examples below also returns same result

+ Using **React** object

render() {

return React.createElement(

'div',

{className: 'App'},

React.createElement('h1', null, 'This is a React App')

);

}

+ Using JSX

render() {

return (

<div className="App">

<h1>This is a React App</h1>

</div>

);

}

**7. JSX Restrictions**

- **class** attribute of element must change to **className**

- value in **return()** only 1 element

**8. Creating a Functional Component**

**- src/Person/Person.js:**

import React from 'react';

const person = () => {

return (<p>I'm a Person!</p>);

}

export default person;

**- src/App.js:**

import Person from './Person/Person';

function App() {

return (

<div className="App">

<h1>This is a React App</h1>

<Person/>

</div>

);

}

**9. Components & JSX Cheat Sheet**

**10. Working with Components & Re-Using Them**

- Focus code in each files and make it much more maintainable

- Easy reusing

**11. Outputting Dynamic Content**

**-** Run Javascript code in JSX:

return (<p>I'm a Person! {Math.random()}</p>);

**12. Working with Props**

- Give attributes for component:

return (

<div className="App">

<h1>This is a React App</h1>

<Person name="Person1" age="30"/>

<Person name="Person2" age="26"/>

<Person name="Person3" age="24"/>

</div>

);

- Take attributes

const person = (props) => {

return (<p>I'm a {props.name} and I am {props.age} years old</p>);

}

- With class-based component, using **this.props**

class Person extends Component {

render() {

return (

<p>I'm a {this.props.name} and I am {this.props.age} years old</p>

);

}

}

**13. Understanding the Children Property**

**-** Get whatever passed in the opening and closing tag of component

<Person name="Person2" age="26">My hobbies: Racing</Person>

- Using **props.children**

const person = (props) => {

return (

<div>

<p>I'm a {props.name} and I am {props.age} years old</p>

<p>{props.children}</p>

</div>

);

}

**14. Understanding and Using State**

**- state** is a special property which is managed from inside a component.

- Declare state in class-based component

state = {

persons: [

{ name: 'Person1', age: 30 },

{ name: 'Person2', age: 26 },

{ name: 'Person3', age: 24 }

]

};

- Using state

render() {

return (

<div className="App">

<h1>This is a React App</h1>

<Person

name={this.state.persons[0].name}

age={this.state.persons[0].age}/>

<Person

name={this.state.persons[1].name}

age={this.state.persons[1].age}>

My hobbies: Racing

</Person>

<Person

name={this.state.persons[2].name}

age={this.state.persons[2].age}/>

</div>

);

}

**15. Props and State**

**16. Handling Events with Methods**

**-** Add event

<button onClick={this.handlerSwitchName}>Switch Name</button>

**-** Declare handler

handlerSwitchName = () => {

console.log('clicked');

};

**17. Events Can Be Listened**

**-** Clipboard events: **onCopy onCut onPaste**

- Composition events: **onCompositionEnd onCompositionStart onCompositionUpdate**

- Keyboard events: **onKeyDown onKeyPress onKeyUp**

- Focus events: **onFocus onBlur**

- Form events: **onChange onInput onInvalid onSubmit**

- Mouse events: **onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave**

- Selection events: **onSelect**

- Touch events: **onTouchCancel onTouchEnd onTouchMove onTouchStart**

- UI events: **onScroll**

- Wheel events: **onWheel**

- Media events: **onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend onTimeUpdate onVolumeChange onWaiting**

- Image events: **onLoad onError**

- Animation events: **onAnimationStart onAnimationEnd onAnimationIteration**

- Transition events: **onTransitionEnd**

- Other events: **onToggle**

**18. Manipulating the State**

**-** Using **setState()** function from **Component** class to update state.

handlerSwitchName = () => {

this.setState({

persons: [

{ name: 'Person1.1', age: 29 },

{ name: 'Person2.1', age: 27 },

{ name: 'Person3.1', age: 25 }

]

})

};

- **setState()** only merges a new state object to an old state object. Thus, other properties will not be updated.

- Only when *state* or *props* change, DOM is re-rendered.

**19. Function Component Naming**

const App = () => { ... }

**20. Using the useState() Hook for State Manipulation**

**-** Import **useState**

import React, { useState } from 'react';

- Initialize state

const [personState, setPersonState] = useState({

persons: [

{ name: 'Person1', age: 28},

{ name: 'Person2', age: 26},

{ name: 'Person3', age: 24}

]

});

- If component has many state object, **useState** can re-use multi time

**const [otherState, setOtherState] = useState({otherState: 'other value'});**

**-** Define function stored in a constant

const handlerSwitchName = () => {

setPersonState({

persons: [

{ name: 'Person1.1', age: 29},

{ name: 'Person2.1', age: 27},

{ name: 'Person3.1', age: 25}

]

});

};

- Return directly JSX instead of **render()**

**21. Stateless vs Stateful Components**

- Stateless components (presentational component, dumb component): component has no internal state management.

- Stateful component (container component, smart component): component has internal state management.

- Use stateless components as much as possible to make applications easy to reuse and maintain.

**22. Passing Method References Between Components**

- A method can be passed as props, so that you can call a method in another component.

<Person

name={this.state.persons[0].name}

age={this.state.persons[0].age}

click={this.handlerSwitchName}/>

- Component uses **handlerSwitchName** method as a property of props.

<p onClick={props.click}>

I'm a {props.name} and I am {props.age} years old

</p>

- To pass arguments to function, using **bind** function.

<Person

name={this.state.persons[0].name}

age={this.state.persons[0].age}

click={this.handlerSwitchName.bind(this, 'Person1.2')}/>

- Or using a function to call the method.

<button onClick={() => this.handlerSwitchName('Person1.1')}>

Switch Name

</button>

- Recommend: Using **bind** instead of function because the function can be inefficient.

**23. Adding Two Way Binding**

**-** Create **changeNameHandler** method

changeNameHandler = (event) => {

this.setState({

persons: [

{ name: 'Person1.1', age: 29 },

{ name: event.target.value, age: 27 },

{ name: 'Person3.1', age: 25 }

]

})

}

- Add **changeNameHandler** to **Person**  component

<Person

name={this.state.persons[1].name}

age={this.state.persons[1].age}

click={this.switchNameHandler.bind(this, 'Person1.3')}

changed={this.changeNameHandler}>

My hobbies: Racing

</Person>

- Add props handler to **Person**  component

<input type="text" onChange={props.changed}/>

**24. Adding Styling with Stylesheets**

- **src/Person/Person.css**

**.Person {**

**width: 60%;**

**margin: 16px auto;**

**border: 1px solid #eee;**

**box-shadow: 0 2px 3px #ccc;**

**padding: 16px;**

**text-align: center;**

**}**

- Import style file

import './Person.css';

- Using class in element

<div className="Person">...</div>

**25. Working with Inline Styles**

**-** Create constant object

const style = {

backgroundColor: 'white',

font: 'inherit',

border: '1px solid blue',

padding: '8px'

};

- Add **style** property for element

<button

style={style}

onClick={() => this.switchNameHandler('Person1.1')}>

Switch Name

</button>

**26. Practicing the Basic Syntax**

**27. Assignment Solution**

**28. Useful Resources and Links**

**IV. Working with Lists and Conditionals**

**1. Module Introduction**

**2. Rendering Content Conditionally**

**-** Add **togglePersonsHandler** method to toggle display person. Using the arrow function to make sure **this** keyword under all circumstances always returns to **App** class.

togglePersonsHandler = () => {

let isShowPersons = this.state.showPersons;

this.setState({

showPersons: !isShowPersons

});

}

- Pass props to **button** element.

<button

style={style}

onClick={this.togglePersonsHandler}>

Switch Name

</button>

- Enclose the **div** element in curly braces and use the ternary expression to check the condition.

{ this.state.showPersons ?

<div>

// Person list

</div> : null

}

**3. Handling Dynamic Content The Javascript Way**

**-** Grap persons code to **persons** variable

let persons = null;

if (this.state.showPersons) {

persons = (

<div>

// Person list

</div>

);

}

- Put persons variable to code in curly braces

<div className="App">

<h1>This is a React App</h1>

<button style={style} onClick={this.togglePersonsHandler}>

Switch Name

</button>

{persons}

</div>

**4. Outputting Lists (Intro)**

**5. Outputting Lists**

**-** Using array convert function map to convert array object to JSX

persons = (

<div>

{this.state.persons.map(person => {

return <Person

name={person.name}

age={person.age}/>

})}

</div>

);

**6. Lists and State**

**-** Add **deletePersonHandler** method to delete person

deletePersonHandler = (index) => {

const persons = this.state.persons;

persons.splice(index, 1);

this.setState({persons: persons});

}

- Add props to **Person** component

<Person

click={() => this.deletePersonHandler(index)}

name={person.name}

age={person.age}/>

**7. Updating State Immutably**

- Getting persons directly from state and splice can lead to unpredictable apps because **'persons'** constantly actually point to the original **'persons'** object.

const persons = this.state.persons;

persons.splice(index, 1);

- Simply copy array by **splice** function

let isShowPersons = this.state.showPersons.splice();

- Or using spread operator

const persons = [...this.state.persons];

**8. Lists and Keys**

**-** React has something called virtual DOM where it compares actual adjustment with the previous DOM it rendered. Thus, react needs a unique **key** prop to specify which element changed. Otherwise, react has to render the whole list.

- Adding **id** property for each person

state = {

persons: [

{ id: 'key1', name: 'Person1', age: 30 },

{ id: 'key2', name: 'Person2', age: 26 },

{ id: 'key3', name: 'Person3', age: 24 }

],

showPersons: false

};

- Passing **key** prop to component

<Person

click={this.deletePersonHandler}

name={person.name}

age={person.age}

key={person.id}/>

**9. Flexible Lists**

**-** Passed **changed** prop to **Person** component

changed={(event) => this.changeNameHandler(event, person.id)}

- Rewrite **changeNameHandler** method

changeNameHandler = (event, id) => {

const persons = [...this.state.persons];

const person = persons.find(p => p.id === id);

person.name = event.target.value;

this.setState({persons: persons});

}

**10. Wrap Up**

**11. Practicing Lists and Conditionals**

**12. Useful Resources and Links**

**V. Styling React Components and Elements**

**1. Module Introduction**

**2. Outlining the Problem Set**

**-** Cannot use pseudo selectors, media queries in inline styles.

- Change styles dynamically

**3. Setting Styles Dynamically**

**-** Declaring style object

const style = {

backgroundColor: 'green',

color: 'white',

font: 'inherit',

border: '1px solid blue',

padding: '8px'

};

- Changing styles by Javascript

if (this.state.showPersons) {

style.backgroundColor = 'red';

}

**4. Setting Class Names Dynamically**

- Adding css class

.red {

color: red;

}

.bold {

font-weight: 700;

}

- Handling styles by javascript

const classes = [];

if (this.state.persons.length <= 2) {

classes.push('red');

}

if (this.state.persons.length <= 1) {

classes.push('bold');

}

- Adding styles class to element

<p className={classes.join(' ')}>This is really working!</p>

**5. Adding and Using Radium**

- Install **radium** package:

npm install radium

- Import package

import Radium from 'radium';

- Wrapping **App** class with **Radium** function which add some extra syntax which will parse styles

export default Radium(App);

- Using pseudo selector **:hover**

const style = {

backgroundColor: 'green',

color: 'white',

font: 'inherit',

border: '1px solid blue',

padding: '8px',

':hover': {

backgroundColor: 'lightgreen',

color: 'black'

}

};

style.backgroundColor = 'red';

style[':hover'] = {

backgroundColor: 'salmon',

color: 'black'

}

**6. Using Radium for Media Queries**

***+*** *Setting up to using media queries*

- Import **StyleRoot** from **radium**

import Radium, { StyleRoot } from 'radium';

- Wrap all application into **<StyleRoot>**

return (

<StyleRoot>

<div className="App">

// Person list

</div>

</StyleRoot>

);

- Styling with media queries

const style = {

'@media (min-width: 1000px)': {

width: '500px'

}

};

**7. Enabling CSS Modules**

**8. Enabling and Using CSS Modules**

***+*** *Enabling CSS Modules*

- **Note 1:** Commit changes before setup

- **Note 2:** This configuration is compatible with **react: ^17.0.1, webpack:4.44.2**

- Run eject **react-script**

npm run eject

- Change CSS configuration in **config/webpack.config.js**

{

test: cssRegex,

exclude: cssModuleRegex,

use: getStyleLoaders({

importLoaders: 1,

sourceMap: isEnvProduction

shouldUseSourceMap

: isEnvDevelopment,

modules: { // add modules options

getLocalIdent: getCSSModuleLocalIdent,

},

}),

sideEffects: true,

},

- Restart develop serve

npm start

*+ Using CSS Modules*

- Import CSS files

import classes from './App.css';

- Using class in css file

<div className={classes.App}>

**9. More on CSS Modules**

**-** To define a global CSS class, prefix the selector with **:global**

**:global .Post {**

**/\* css property \*/**

**}**

**10. Adding Pseudo Selectors**

- Add styles

.App button {

background-color: green;

color: white;

font: inherit;

border: 1px solid blue;

padding: 8px;

}

.App button:hover {

background-color: lightgreen;

color: black;

}

.App button.Red {

background-color: red;

}

.App button.Red:hover {

background-color: salmon;

color: black;

}

- Use CSS Modules to get styles class

let buttonClass = '';

buttonClass = classes.Red;

- Add class to element

<button

className={buttonClass}

onClick={this.togglePersonsHandler}>

Toggle Name

</button>

**11. Working with Media Queries**

- Add styles

@media (min-width: 800px) {

.Person {

width: 400px;

}

}

- Add class to element

<div className={classes.Person}></div>

- The CSS Modules recognizes styles class automatically even it wrapped in media queries

**12. Useful Resources and Links**

**VI. Debugging React Apps**

**1. Module Introduction**

**2. Understanding Errors Messages**

**3. Finding Logical Errors by Using Dev Tools and Sourcemaps**

**4. Working with the React Developer Tools**

**5. Using Error Boundaries (React 16+)**

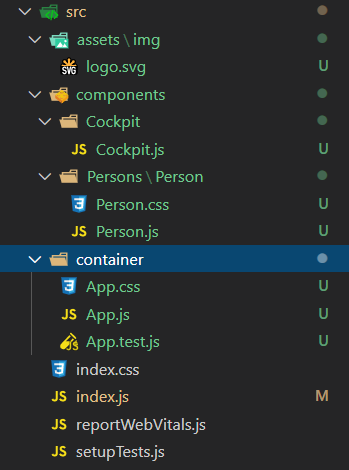
**6. Wrap Up**

**7. Useful Resources and Links**

**VII. Diving Deeper into Components and React Internals**

**1. Module Introduction**

**2. A Better Project Structure**

****

**3. Splitting an App into Components**

**-** Create **Persons** component in **/components/Persons/Persons.js**

**import React from 'react';**

**import Person from './Person/Person';**

**const persons = (props) => props.persons.map((person, index) => {**

**return <Person**

**click={() => props.clicked(index)}**

**changed={(event) => props.changed(event, person.id)}**

**name={person.name}**

**age={person.age}**

**key={person.id}/>**

**});**

**export default persons;**

**-** Create **Cockpit** component in **/components/Cockpit/Cockpit.js**

**import React from 'react';**

**import classes from './Cockpit.css';**

**const cockpit = (props) => {**

**const textClasses = [];**

**let buttonClass = '';**

**if (props.showPersons) {**

**buttonClass = classes.Red;**

**}**

**if (props.persons.length <= 2) {**

**textClasses.push(classes.red);**

**}**

**if (props.persons.length <= 1) {**

**textClasses.push(classes.bold);**

**}**

**return (**

**<div className={classes.Cockpit}>**

**<h1>This is a React App</h1>**

**<p className={textClasses.join(' ')}>This is really working!</p>**

**<button**

**className={buttonClass}**

**onClick={props.clicked}>**

**Toggle Name**

**</button>**

**</div>**

**);**

**};**

**export default cockpit;**

**-** Add styles for component in **/components/Cockpit/Cockpit.css**

**.red {**

**color: red;**

**}**

**.bold {**

**font-weight: 700;**

**}**

**.Cockpit button {**

**background-color: green;**

**color: white;**

**font: inherit;**

**border: 1px solid blue;**

**padding: 8px;**

**}**

**.Cockpit button:hover {**

**background-color: lightgreen;**

**color: black;**

**}**

**.Cockpit button.Red {**

**background-color: red;**

**}**

**.Cockpit button.Red:hover {**

**background-color: salmon;**

**color: black;**

**}**

- Refactor **App.js**

+ Import components:

import Cockpit from '../components/Cockpit/Cockpit';

import Persons from '../components/Persons/Persons';

+ Refactor components:

render() {

let persons = null;

if (this.state.showPersons) {

persons = (

<Persons

persons={this.state.persons}

clicked={this.deletePersonHandler}

changed={this.changeNameHandler} />

);

}

return (

<div className={classes.App}>

<Cockpit

showPersons={this.state.showPersons}

persons={this.state.persons}

clicked={this.togglePersonsHandler} />

{persons}

</div>

);

}

**4. Comparing Stateless and Stateful Components**

- Using as many stateless components as possible.

**5. Class-based vs Functional Components**

| **Class-based** | **Functional** |
| --- | --- |
| **-** class XY extends Component  - Can access to State  - Can use Lifecycle Hooks  - Access State and Props via “this”  - Use if you need to manage State or access Lifecycle Hooks and you don’t want to use React Hooks | **-** const XY = props => {...}  - Can access to State (useState())  - Cannot use Lifecycle Hooks  - Access Props via ‘props’  - Use in all other Cases |

**6. Class Component Lifecycle Overview**

**7. Component Creation Lifecycle in Action**

**-** constructor(props)

-> getDerivedStateFromProps(props, state)

-> render()

-> Render Child Component

-> componentDidMount() (Side-effects)

**8. Component Update Lifecycle (for props changes)**

**-** getDerivedStateFromProps(props, state)

-> shouldComponentUpdate(nextProps, nextState)

-> render()

-> Render Child Component

-> getSnapshotBeforeUpdate(prevProps, prevState()

-> componentDidUpdate() (Side-effects)

**9. Component Update Lifecycle (for state changes)**

**-** componentDidMount()

-> shouldComponentUpdate(nextProps, nextState)

-> componentDidUpdate()

**10. Using useEffect()**

**-** useEffect()is a React hook

- useEffect()runs everytime DOM is re-rendered.

**11. Controlling the useEffect() Behavior**

**-** variables point all the variables or all the data that actually are used in your effect, put second argument to useEffect()

**-** In Cockpit.js

**useEffect(() => {**

***console*.log('[Cockpit.js] useEffect');**

**setTimeout(() => {**

**alert('Saved data to cloud!');**

**}, 1000);**

**}, []);**

**12. Cleaning up with Lifecycle Hook & useEffect()**

**-** componentWillUnmount() runs right before the component is removed.

- Return an anonymous function in useEffect() that will run after every render cycle.

return () => {

*console*.log('[Cockpit.js] cleanup work in useEffect');

};

- Add button to remove cockpit in app.js

<button *onClick*={() => {*this*.setState({ showCockpit: false })}}>

Remove Cockpit

</button>

{*this*.state.showCockpit ? (

<*Cockpit*

*title*={*this*.props.appTitle}

*showPersons*={*this*.state.showPersons}

*persons*={*this*.state.persons}

*clicked*={*this*.togglePersonsHandler} />

) : null}

{persons}

- In case you have some operation which actually should be canceled whenever the component re-rendered.

useEffect(() => {

*console*.log('[Cockpit.js] 2nd useEffect');

return () => {

*console*.log('[Cockpit.js] cleanup work in 2nd useEffect');

};

});

**13. Cleaning up with Lifecycle Hooks & useEffect()**

**14. Using shouldComponentUpdate for Optimization**

**-** In Persons.js

shouldComponentUpdate(*nextProps*, *nextState*) {

*console*.log('[Persons.js] should ComponentUpdate');

if (nextProps.persons !== *this*.props.persons) {

return true;

} else {

return false;

}

}

**15. Optimizing Functional Components with React.memo()**

**-** React.memo() stores a snapshot of the component and only re-render if its input changes

export default React.memo(Cockpit);

- Change cockpit component

<*Cockpit*

*title*={*this*.props.appTitle}

*showPersons*={*this*.state.showPersons}

*personsLength*={*this*.state.persons.length}

*clicked*={*this*.togglePersonsHandler} />

**16. When should you optimize**

**-** Don’t use shouldComponentUpdate and React.memo()when all or almost all cases where parent components update, children components need updating too.

**17. PureComponent instead of shouldComponentUpdate**

**-** PureComponent is a normal Component that is already implemented with a complete props check.

class Persons extends *PureComponent* {

**18. How React Updates the DOM**

**-** React stores 2 versions of Virtual DOM: Old Virtual DOM and Re-rendered Virtual DOM

- When render() is called, React compares the difference between 2 versions of virtual DOM.

- If there are changes between 2 versions of virtual DOM, React changes differences into real DOM. Otherwise, React does not change anything.

**19. Rendering Adjacent JSX Elements**

There are2 ways to render adjacent JSX Elements

- Using array:

render() {

*console*.log('[Person.js] rendering...');

return [

<p *key*="i1" *onClick*={*this*.props.click}>

I'm a {*this*.props.name} and I am {*this*.props.age} years old

</p>,

<p *key*="i2">{*this*.props.children}</p>,

<input *key*="i3" *type*="text" *onChange*={*this*.props.changed}/>,

];

}

- Create a wrapping component:

+ Create hoc/auxiliary.js

const aux = *props* => props.children;

export default aux;

+ Wrapping elements:

<*Aux*>

<p *onClick*={*this*.props.click}>

I'm a {*this*.props.name} and I am {*this*.props.age} years old

</p>

<p>{*this*.props.children}</p>

<input *type*="text" *onChange*={*this*.props.changed}/>

</*Aux*>

**20. Note for Windows users**

**21. Using React.Fragment**

**-** React.Fragment does exactly the same things a auxiliary component

**<*React.Fragment*>**

**<p *onClick*={*this*.props.click}>**

**I'm a {*this*.props.name} and I am {*this*.props.age} years old**

**</p>**

**<p>{*this*.props.children}</p>**

**<input *type*="text" *onChange*={*this*.props.changed}/>**

**</*React.Fragment*>**

**22. Higher Order Component (HOC) - Introduction**

**-** Higher Order Component is a component that wraps another component. It does not contain its own logic, styling or add any structure to the JSX code or to the real DOM that will be rendered.

- Create a HOC:

import React from 'react';

const withClass = *props* => (

<div *className*={props.classes}>{props.children}</div>

);

export default withClass;

- Using HOC:

import classes from './App.css';

import WithClass from '../hoc/WithClass';

...

<*WithClass* *classes*={classes.App}>

...

</*WithClass*>

**23. Another From of HOCs**

**-** In withClass.js

const withClass = (*WrappedComponent*, *className*) => {

return *props* => (

<div *className*={className}>

<*WrappedComponent*/>

</div>

);

};

- Using HOC:

export default withClass(App, classes.App);

-> This HOC makes it clear that they are not so much involved in the JSX code that gets rendered but in the logic that runs.

**24. Passing Unknown Props**

- In withClass.js

**const withClass = (*WrappedComponent*, *className*) => {**

**return *props* => (**

**<div *className*={*className*}>**

**<*WrappedComponent* {...props} />**

**</div>**

**);**

**};**

**25. Setting State Correctly**

- To set new state depend on old state:

*this*.setState((*prevState*, *props*) => {

return {

persons: persons,

changeCounter: *prevState*.changeCounter + 1

};

});

**26. Using PropTypes**

- Install prop-types package:

npm install prop-types

- Import package:

import PropTypes from 'prop-types';

- Set up package: add before export module.

Person.propTypes = {

click: PropTypes.func,

name: PropTypes.string,

age: PropTypes.number,

changed: PropTypes.func

};

**27. Using Refs**

- Using function

**+** Add ref keyword into input component:

<input

*type*="text"

*ref*={(*inputEl* => {*this*.inputElement = *inputEl*})}

*onChange*={*this*.props.changed}

*value*={*this*.props.name}

/>

+ Call focus() method:

componentDidMount() {

*this*.inputElement.focus();

}

- Using React.createRef()

+ Create reference object:

constructor(*props*) {

*super*(*props*);

*this*.inputElementRef = *React*.createRef();

}

**+** Add ref keyword into input component:

<input

*type*="text"

*ref*={*this*.inputElementRef}

*onChange*={*this*.props.changed}

*value*={*this*.props.name}

/>

+ Call focus() method:

componentDidMount() {

*this*.inputElementRef.current.focus();

}

**28. Refs with React Hooks**

**-** Using ref keyword in functional component

+ Declare reference:

const toggleBtnRef = useRef(null);

**+** Add ref keyword into button component:

<button

*ref*={toggleBtnRef}

*className*={buttonClass}

*onClick*={*props*.clicked}>

Toggle Name

</button>

+ Call click() method:

useEffect(() => {

console.log('[Cockpit.js] useEffect');

toggleBtnRef.current.click();

return () => {

console.log('[Cockpit.js] cleanup work in useEffect');

};

}, []);

**29. Understanding Prop Chain Problems**

**-** Context was introduced by React and it helps us handle problems that you don’t want to pass state across multiple layers from component A at the top to component B at the bottom when components B, C between don’t really care about it.

**30. Using Context API**

**-** Create ./src/context/auth-context.js:

import React from 'react';

const authContext = *React*.createContext({

authenticated: false,

login: () => {}

});

export default authContext;

- Import context:

import AuthContext from '../../context/auth-context';

- Define loginHandler:

loginHandler = () => {

*this*.setState({authenticated: true});

}

- In App.js, wrap Cockpit component by context provider:

<*AuthContext.Provider*

*value*={{

authenticated: *this*.state.authenticated,

login: *this*.loginHandler

}}

>

{*this*.state.showCockpit ? (

<*Cockpit*

*title*={*this*.props.appTitle}

*showPersons*={*this*.state.showPersons}

*personsLength*={*this*.state.persons.length}

*clicked*={*this*.togglePersonsHandler} />

) : null}

{persons}

</*AuthContext.Provider*>

- Consume context with authentication status text

<*AuthContext.Consumer*>

{(*context*) =>

*context*.authenticated ? <p>Authenticated!</p> : <p>Please login!</p>

}

</*AuthContext.Consumer*>

- Consume context with login button:

<*AuthContext.Consumer*>

{(*context*) => <button *onClick*={*context*.login}> Login </button>}

</*AuthContext.Consumer*>

**31. contextType & useContext()**

- Using contextType in class-based component:

+ Declare static variable:

static contextType = AuthContext;

+ Use context to get property:

{*this*.context.authenticated ? (<p>Authenticated!</p>) : (<p>Please login!</p>)}

- Using useContext() in functional component:

+ Declare constant with React hook:

const authContext = useContext(AuthContext);

+ Use context to get property:

<button *onClick*={authContext.login}> Login </button>

**32. Wrap Up**

**VIII. A Real App The Burger Builder (Basic Version)**

**1.. About React Hooks**

**2. Module Introduction**

**3. Planning an App in React - Core Steps**

- Component Tree/ Component Structure

- Application State (Data)

- Components vs Containers

**4. Planning an App in React - Layouts and Component Tree**

****

**5. Planning the State**

**6. Setting up the Project**

**-** Enabling CSS Modules

- Use font: ***open sans***

**7. Creating a Layout Component**

**8. Starting Implementation of The Burger Builder Container**

**9. Adding a Dynamic Ingredient Component**

**10. Adding Prop Type Validation**

- Install prop-types package

**11. Starting the Burger Component**

**12. Outputting Burger Ingredients Dynamically**

**13. Calculating the Ingredient Sum Dynamically**

**14. Adding the Build Control Component**

**15. Outputting Multiple Build Controls**

**16. Connecting State to Build Controls**

**17. Removing Ingredients Safely**

**18. Displaying and Updating the Burger Price**

**19. Adding the Order Button**

**20. Creating the Order Summary Modal**

**21. Showing & Hiding the Modal (with Animation!)**

**22. Implementing the Backdrop Component**

**23. Adding a Custom Button Component**

**24. Implementing the Button Component**

**25. Adding the Price to the Order Summary**

**26. Adding a Toolbar**

**27. Using a Logo in our Application**

**28. Adding Reusable Navigation Items**

**29. Creating a Responsive Sidedrawer**

**30. Working on Responsive Adjustments**

**31. More about Responsive Adjustments**

**32. Reusing the Backdrop**

**33. Adding a SideDrawer Toggle Button**

**34. Adding a Hamburger Icon**

**35. Improving the App - Introduction**

**36. Prop Type Validation**

**37. Improving Performance**

**38. Using Component Lifecycle Methods**

**39. Changing the Folder Structure**

**40. Wrap Up**

**IX. Reaching out to the Web (HTTP Ajax)**

**1. Module Introduction**

**2. Understanding Http Requests in React**

**3. Understanding our Project and Introducing Axios**

**-** Mock JSON data: <https://jsonplaceholder.typicode.com/>

- Install axios package:

npm install axios

**4. Creating a Http Request to GET Data**

**5. Rendering Fetched Data to the Screen**

**6. Transforming Data**

**7. Making a Post Selectable**

**8. Fetching Data on Update (without Creating Infinite Loops)**

**9. POSTing Data to the Server**

**10. Sending a DELETE Request**

**11. Fixing a Bug**

**12. Handling Errors Locally**

**13. Adding Interceptors to Execute Code Globally**

**14. Removing Interceptors**

**15. Setting a Default Global Configuration for Axios**

**16. Creating and Using Axios Instances**

**17. Wrap Up**

**X. Burger Builder Project Accessing a Server**

**1. Module Introduction**

**2. Firebase & The Right Database**

**3. Creating the Firebase Project**

**-** Create real-time database on firebase

**4. Creating the Axios Instance**

**5. Sending a POST Request**

**6. Displaying a Spinner while Sending a Request**

**-** Spinner CSS: <https://projects.lukehaas.me/css-loaders/>

**7. Handling Errors**

**8. Retrieving Data from the Backend**

**9. Removing Old Interceptors**

**XI. Multi-Page-Feeling in a Single-Page-App Routing**

**1. Module Introduction**

**2. Routing and SPAs**

****

**3. Setting Up Links**

**4. Setting Up the Router Package**

**5. react-router vs react-router-dom**

**6. Preparing the Project For Routing**

**7. Setting Up and Rendering Routes**

**8. Rendering Components for Routes**

**9. Switching Between Pages**

**10. Using Links to Switch Pages**

**11. Using Routing-Related Props**

**12. The withRouter HOC & Route Props**

**-** Import HOC:

**import { withRouter } from 'react-router-dom';**

**-** Wrap HOC to component:

export default withRouter(post);

**13. Absolute vs Relative Paths**

**-** Build a relative path:

pathname: *this*.props.match.url + '/new-post',

**14. Absolute vs Relative Paths (Article)**

**15. Styling the Active Route**

**16. Passing Route Parameters**

**17. Extracting Route Parameters**

**18. Parsing Query Parameters & the Fragment**

**19. Using Switch to Load a Single Route**

**20. Navigating Programmatically**

**21. Additional Information Regarding Active Links**

**22. Understanding Nested Routes**

**23. Creating Dynamic Nested Routes**

**24. Redirecting Requests**

**25. Conditional Redirects**

**26. Using the History Prop to Redirect (Replace)**

**27. Working with Guards**

**28. Handling the 404 Case (Unknown Routes)**

**29. Loading Routes Lazily**

**30. Lazy Loading with React Suspense (16.6)**

**-** Work with React 16.6 or higher

- Import Suspense from react

import React, { Component, Suspense } from 'react';

- Put component to lazy() method

const Posts = *React*.lazy(() => import('./Posts/Posts'));

- Load component lazily

{*this*.state.showPosts

? (<*Suspense* *fallback*={<div>Loading...</div>}><*Posts*/></*Suspense*>)

: <*NewPost*/>}

**31. Routing and Server Deployment**

**-** When server is set to a subdomain, need to specific basename in BrowserRouter component

<*BrowserRouter* *basename*="/my-app">

**32. Time to Practice - Routing**

**33. Wrap Up**

**34. Useful Resources & Links**

**XII. Adding Routing to our Burger Project**

**1. Module Introduction**

**2. Building the Checkout Container**

**3. Setting Up Routing & Routes**

**4. Navigating to the Checkout Page**

**5. Navigating Back & To Next Page**

**6. Passing Ingredients via Query Params**

**7. Navigating to the Contact Data Component**

**8. Order Submission & Passing Data Between Pages**

**9. Adding an Orders Page**

**10. Implementing Navigation Links**

**11. Fetching Orders**

**12. Outputting the Orders**

**13. Wrap Up**

**XIII. Forms and Form Validation**

**1. Module Introduction**

**2. Analyzing the App**

**3. Creating a Custom Dynamic Input Component**

**4. Setting Up a JS Config for the Form**

**5. Dynamically Create Inputs based on JS Config**

**6. Adding a Dropdown Component**

**7. Handling User Input**

**8. Handling Form Submission**

**9. Adding Custom Form Validation**

- Alternatives to the manual approach taken in this course:

+ *react-validation package*: <https://www.npmjs.com/package/react-validation>

*+ formsy-react package:* <https://github.com/christianalfoni/formsy-react>

**10. Fixing a Common Validation Gotcha**

**11. Adding Validation Feedback**

**12. Improving Visual Feedback**

**13. Showing Error Messages**

**14. Handling Overall Form Validity**

**15. Working on an Error**

**16. Fixing a Bug**

**XIV. Redux**

**1. Module Introduction**

**2. Understanding State**

**3. The Complexity of Managing State**

**4. Understanding the Redux Flow**

**-** Redux is a third party library which totally independent of react

- Redux has a clearly defined process of how your state may change

- Redux flow



**5. Setting Up Reducer and Store**

**6. Dispatching Actions**

**7. Adding Subscriptions**

**8. Connecting React to Redux**

**9. Connecting the Store to React**

**10. Dispatching Actions from within the Component**

**11. Time to Practice - Dispatching Actions**

**12. Passing and Retrieving Data with Action**

**13. Switch-Case in the Reducer**

**14. Updating State Immutably**

**15. Updating Arrays Immutably**

**16. Immutable Update Patterns**

**17. Outsourcing Action Types**

**18. Combining Multiple Reducers**

**19. Understanding State Types**

| **Type** | **Example** | **Use Redux** |
| --- | --- | --- |
| Local UI State | Show/ Hide Backdrop | Mostly handled within components |
| Persistent State | All Users, all Posts,.. | Stored on server, relevant slice managed by Redux |
| Client State | Is authenticated? Filter set by User,... | Managed via Redux |

**20. Time to Practice - Redux Basics**

**21. Combining Local UI State and Redux**

**22. Wrap Up**

**XV. Adding Redux to our Project**

**1. Module Introduction**

**2. Installing Redux and React Redux**

**3. Basic Redux Setup**

**4. Finishing the Reducer for Ingredients**

**5. Connecting the Burger Builder Container to our Store**

**6. Working on the Total Price Calculation**

**7. Redux & UI State**

**8. Adjusting Checkout and Contact Data**

**9. Wrap Up**

**XVI. Redux Advanced**

**1. Module Introduction**

**2. Adding Middleware**

**-** Middleware is a term used for functions or the code in general. Middleware hooks into a process which then gets executed as part of that process without stopping it.

**3. Using the Redux Devtools**

- Redux Devtools:<https://github.com/zalmoxisus/redux-devtools-extension>

**4. Executing Asynchronous Code - Introduction**

**5. Introducing Action Creators**

**6. Action Creators & Async Code**

**7. Handling Asynchronous Code**

**8. Restructuring Actions**

**9. Where to Put Data Transforming Logic**

**-** Logic to transform state can be put in **Action Creators** and **Reducer**

| **Action Creators** | **Reducer** |
| --- | --- |
| Can run Async Code | Pure, Sync Code Only |
| Shouldn’t Prepare the State Update Too Much | Core Redux Concept: Reducers update the State |

**10. Using Action Creators and Get State**

**11. Using Utility Functions**

**12. A Leaner Switch Case Statement**

**13. An Alternative Folder Structure**

**14. Diving Much Deeper**

- Redux document:<https://redux.js.org/usage/index>

**15. Wrap Up**

**XVII. Redux Advanced Burger Project**

**1. Module Introduction**

**2. Installing the Redux Devtools**

**3. Preparing the Folder Structure**

**4. Creating Action Creators**

**5. Executing Asynchronous Code**

**6. Fetching Ingredients Asynchronously**

**7. Initializing Ingredients in the BurgerBuilder**

**8. Changing the Order of our Ingredients Manually**

**9. Adding Order Actions**

**10. Connecting Contact Data Container & Actions**

**11. The Order Reducer**

**12. Working on Order Actions**

**13. Redirect to Improve UX**

**14. Combining Reducers**

**15. Handling Purchases & Updating the UI**

**16. Resetting the Price after Purchases**

**17. Fetching Orders (via Redux)**

**18. Checking our Implemented Functionalities**

**19. Refactoring Reducers**

**20. Refactoring Reducers Continued**

**21. Wrap Up**

**XVIII. Redux Advanced Burger Project**

**1. Module Introduction**

**2. Understanding Authentication in Single Page Applications**

- Use local storage to store token

**3. Required App Adjustments**

**4. Adding an Auth Form**

**5. Adding Actions**

**6. Getting a Token from the Backend**

- Document: <https://firebase.google.com/docs/reference/rest/auth#section-create-email-password>

**7. Adding Sign-In**

**8. Storing the Token**

**9. Adding a Spinner**

**10. Logging Users Out**

**11. Accessing Protected Resources**

**12. Updating the UI Depending on Auth State**

**13. Adding a Logout Link**

**14. Forwarding Unauthenticated Users**

**15. Redirecting the User to the Checkout Page**

**16. Persistent Auth State with localStorage**

**17. Fixing Connect + Routing Errors**

**18. Ensuring App Security**

**19. Guarding Routes**

**20. Displaying User Specific Orders**

{

"rules": {

"ingredients": {

".read": true,

".write": true,

},

"orders": {

".read": "auth != null",

".write": "auth != null",

".indexOn": ["userId"]

}

}

}

**21. Wrap Up**

**XIX. Improving our Burger Project**

**1. Module Introduction**

**2. Fixing the Redirect to the Frontpage**

**3. Using updateObject in the Entire App**

**4. Sharing the Validation Method**

**5. Using Environment Variables**

**6. Removing console.log()s**

**7. Adding Lazy Loading**

**8. Wrap Up**

**XX. Testing**

**1. Module Introduction**

**2. What is Testing**

**3. Required Testing Tools**

**-** Test Runner: executes tests and provides validation library (Jest)

- Testing Utilities: simulates the React App (mounts components, allow you to dig into DOM) (Enzyme)

**4. What To Test**

**-** Don’t test the library

- Don’t test complex connections

- Do test isolated units

- Do test your conditional outputs

**5. Writing our First Test**

- Install enzyme, react-test-renderer, enzyme-adapter-react-16 packages:

npm install enzyme react-test-renderer enzyme-adapter-react-16

- Install jest-circus packages:

npm install jest-circus

- Change testRunner configuration in packages.json :

**"testRunner": "<rootDir>\\node\_modules\\jest-circus\\runner.js",**

**6. Testing Components Continued**

**7. Jest and Enzyme Documentations**

- *Enzyme API*: <http://airbnb.io/enzyme/docs/api/>

- *Jest Docs*: <https://facebook.github.io/jest/>

**8. Testing Components Correctly**

**9. Testing Containers**

**10. How to Test Redux**

**11. Wrap Up**

**XXI. Deploying the App to the Web**

**1. Module Introduction**

**2. Deployment Steps**

**-** Check (& Adjust) Basepath:

<*BrowserRouter* *basename*="/my-app/">

- Build & Optimize Project

npm run build

- Server must always serve index.html (also for 404 cases)

- Upload build Artifacts to (static) Server

**3. Building the Project**

- Build & Optimize Project

npm run build

**4. Example Deploying on Firebase**

- Install firebase tools:

npm install -g firebase-tools

- Login to google:

firebase login

- Initiate project

firebase init

- Deploy website

firebase deploy

**5. Wrap Up**

**XXII. Bonus Working with Webpack**

**1. Module Introduction**

**2. Important Use Webpack 3**

**3. Introducing Webpack**

**4. How Webpack works**

****

**5. Basic Workflow Requirements**

**-** Compile Net-Gen Javascript Features

- Handle JSX

- CSS Autoprefixing

- Support Image Imports

- Optimize code

**6. Project & npm Setup**

**-** Initiate project:

npm init

- Install webpack:

npm install --save-dev webpack webpack-dev-server

- Install webpack-cli:

npm install --save-dev webpack-cli

**7. Creating a Basic Folder & File Structure**

**8. Creating the Basic React Application**

**9. Installing Production Dependencies**

- Install dependencies:

npm install --save-dev react react-dom react-router-dom

- **Note:** react-router-dom version 6 is not compatible with this application, use version 5 instead.

**10. Setting Up the Basic Webpack Config**

**11. Adding File Rules**

**12. Introducing Babel**

- Install babel:

npm install --save-dev babel-loader @babel/core @babel/preset-react @babel/preset-env

**- Note:** Webpack 5 is using @babel/core, @babel/preset-react, @babel/preset-env instead of babel-core, babel-preset-react, babel-preset-env

**13. Adding CSS File Support**

- Install css loader:

npm install --save-dev css-loader style-loader postcss-loader autoprefixer

**14. Creating Rules for Images**

- Install image/file loader:

npm install --save-dev url-loader file-loader

**15. Lazy Loading**

- Install lazy loading:

npm install --save-dev @babel/plugin-syntax-dynamic-import @babel/preset-stage-2

**- Note:** Webpack 5 is using babel-plugin-syntax-dynamic-import, babel-preset-2 instead of @babel/plugin-syntax-dynamic-import, @babel/preset-stage-2.

**16. Injecting the Script into the index.html File**

- Install package:

npm install --save-dev html-webpack-plugin

**17. Creating the Production Workflow**

- Install package:

npm install --save-dev rimraf

**18. Wrap Up**

**19. Adding babel-polyfill**

**20. Useful Resources & Links**

- *Webpack:* <https://webpack.js.org/concepts/>

- *Babel*: <https://babeljs.io/>

**XXIII. Bonus Next.js**

**1. Module Introduction**

**2. Understanding Server Side Rendering**

**3. Setting Up a Project**

- Install package:

npm install next react react-dom

**4. Understanding the Basics**

**5. Next.js & Components & Pages**

**6. Styling our App in Next.js**

**7. Handling (404) Errors**

**8. A Special Lifecycle Hook**

**9. Deploying our App**

**10. Useful Resources & Links**

- *Github*: <https://github.com/vercel/next.js>

**XXIV. Bonus Animations in React Apps**

**1. Module Introduction**

**2. Preparing the Demo Project**

**3. Using CSS Transitions**

**4. Using CSS Animations**

**5. CSS Transition & Animations Limitations**

**6. Using ReactTransitionGroup**

- Install react transition group

npm install react-transition-group

**7. Using the Transition Component**

**8. Wrapping the Transition Component**

**9. Animation Timings**

**10. Transition Events**

**11. The CSSTransition Component**

**12. Customizing CSS Classnames**

**13. Animating Lists**

**14. Alternative Animation Packages**

- *React motion*: <https://www.npmjs.com/package/react-motion>

- *React move*: <https://www.npmjs.com/package/react-move>

- *React router transition*: <https://www.npmjs.com/package/react-router-transition>

**15. Wrap Up**

**XXV. Bonus A Brief Introduction to Redux Saga**

**1. Module Introduction**

**2. Installing Redux Saga**

- Install react saga

npm install redux-saga

**3. Creating our First Saga**

**4. Hooking the Saga Up (to the Store and Actions)**

**5. Moving Logic from the Action Creator to a Saga**

**6. Moving More Logic Into Sagas**

**7. Handling Authentication with a Saga**

**8. Handling Auto-Sign-In with a Saga**

**9. Moving the BurgerBuilder Side Effects into a Saga**

**10. Moving the Orders Side Effects into Sagas**

**11. Why Sagas can be Helpful**

**12. Diving Deeper into Sagas**

**13. Useful Resources & Links**

- *Redux Saga: Full Documentation:*<https://redux-saga.js.org/>

- *Advanced Concepts:* <https://redux-saga.js.org/docs/advanced/>

- *API Reference:* <https://redux-saga.js.org/docs/api/>

- *Pros & Cons for Redux Saga vs Thunks*: <https://stackoverflow.com/questions/34930735/pros-cons-of-using-redux-saga-with-es6-generators-vs-redux-thunk-with-es2017-asy/34933395>

**XXVI. React Hooks**

**1. Introduction**

**2. What are Hooks**

**3. Enabling Hooks**

- Install react version 16.8 or above to include react hook:

npm install

**4. The useState() Hook**

**5. Adding Array Destructuring**

**6. Using Multiple State**

**7. Using One State Instead**

**8. The Rules of Hooks**

- useState() must be called at the root level of component function

**9. Sending Data via Http**

**10. The useEffect() Hook**

**11. Controlling Effect Execution**

**12. Effect Cleanup**

**13. Converting the App Component**

**14. The useContext() Hook**

**15. State and Effects Gotchas**

**17. useReducer() vs useState()**

**18. Working with References and useRef()**

**19. Preparing & Optimizing**

**20. Avoiding Unnecessary Re-Rendering**

**21. How to think about Functional Components with Hooks**

**22. Creating a Custom Hook**

**23. Wrap Up**

**XXVII. Using Hooks in the Burger Builder**

**1. Introduction**

**2. Converting App**

**3. Routing with React.lazy()**

**4. Converting the Layout Component**

**5. Converting withErrorHandler HOC**

**6. Adjusting the Order & Checkout Containers**

**7. Add Hooks to ContactData**

**8. Converting the BurgerBuilder Container**

**9. Adjusting Auth & Logout Components**

**10. Using React.memo() & More!**

**11. Adding a Custom Error Handling Hook**

**12. Wrap Up**

**XXVIII. Bonus Building the Burger CSS**

**1. Building the Burger CSS Code**

- *Burger CSS*: <https://codepen.io/anon/pen/PJrgga>

**XXIV. Next Steps and Course Roundup**

**1. Module Introduction**

**2. React Rocks! Where to find more Examples**

**-** *React rock*: <https://react.rocks/>

**3. More Inspiration Real-World Projects Built with React**

**4. Static React Apps with Gatsby.js**

**-** *GasbyJs*: <https://www.gatsbyjs.com/>

**5. Introducing React Native**

**-***React Native:* <https://reactnative.dev/>

**6. Component Libraries (Example Material UI)**

**-***Material UI*: <https://mui.com/>

- *Bootstrap React:*<https://react-bootstrap.github.io/>

**7. Smaller Apps with Preact**

**-** *Preact:* <https://preactjs.com/>

**8. Comparing React with Preact**

- *JS framework benchmark*: <https://github.com/krausest/js-framework-benchmark>

**9. Congratulations**

**10. Bonus More Content!**

- *More course*: <https://academind.com/learn/our-courses/>