**REACT JS**

Codesandbox.io -> for practice basics and some prblms

Html global attributes in w3schools - <https://www.w3schools.com/tags/ref_standardattributes.asp>

<https://picsum.photos/> -> for random images

**installing react js:**

1. Check node is up-to-date (node –version in hyper/terminal)
2. Install vs code
3. Create react app

<https://reactjs.org/docs/create-a-new-react-app.html>

npx create-react-app my-app

cd my-app

npm start

If stuck -> <https://stackoverflow.com/questions/59188624/template-not-provided-using-create-react-app>

1. Run app

cd /c/Users/ksanj/my-app

then

=> npm start

<https://flatuicolors.com/> - for different colors

<https://material-ui.com/components/material-icons/#material-icons> – for material icons and designs for react components… similar to bootstrap

<https://www.transparenttextures.com/> - for background texturesss

Installing reactjs

npm install create-react-app -g // -g means globally

create-react-app myapp –scripts-version 1.1.5 // to create a react app with myapp name - // and scripts for folder structure.

**Guide:**

**SECTION 3: UNDERSTANDING THE BASE FEATURES & SYNTAX:**

Components can be created in two ways:

Class based component:

* Class will extends Component from react module.
* It uses render() to render jsx
* We should use this keyword to access the properties of the class
* We cannot use var, const variables inside class else we can use varibles directly withour declaring var, const, let etc.
* here we use variables which is properties
* state is managed from inside the component while props are mangaed from outside
* state is a special keyword to hold state values
* setState() => {}, is used to set the values of the state object

Example Code:

import React, { Component } from "react";

import "./App.css";

import Person from "./Person/Person";

class App extends Component {

  // here we use variables which is properties

  // state is managed from inside the component while props are mangaed from outside

  // and works only in class based component

  state = {

    persons: [

      { name: "sanjay", age: "23" },

      { name: "bujji", age: "22" },

      { name: "pintu", age: "15" },

    ],

    otherState: "some other value",

  };

  switchNameHandler = () => {

    // console.log("Was clicked!!!");

    // DONT DO THIS: this.state.persons[0].name = 'sanjay kumar';

    this.setState({

      persons: [

        { name: "sanjay kumar", age: "23" },

        { name: "bujji", age: "22" },

        { name: "pintu", age: "16" },

      ],

    });

  };

  render() {

    return (

      <div className="App">

        <h1>Hi, I'm a react app./</h1>

        {/\* onClick={this.switchNameHandler()} => if put paranthesis it will immediately cal, we nly pas refernce \*/}

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

        <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 hobby: Car racing

        </Person>

        <Person

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

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

        />

      </div>

    );

    // return React.createElement(

    //   "div",

    //   { className: 'App' },

    //   React.createElement("h1", null, "Hi, I'm a react app!!!")

    // );

  }

}

44. Function based component:

* It is just like a function that returns jsx code and no need of render() here.
* useState()

example code:

import React, { useState } from "react";

import "./App.css";

import Person from "./Person/Person";

const app = (props) => {

  const [personsState, setPersonsState] = useState({

    persons: [

      { name: "sanjay", age: "23" },

      { name: "bujji", age: "22" },

      { name: "pintu", age: "15" },

    ],

  });

  const [otherState, setOtherState] = useState({

    otherState: "some other value",

  });

  const switchNameHandler = () => {

    // console.log("Was clicked!!!");

    // DONT DO THIS: this.state.persons[0].name = 'sanjay kumar';

    setPersonsState({

      persons: [

        { name: "sanjay kumar", age: "23" },

        { name: "bujji", age: "22" },

        { name: "pintu", age: "16" },

      ],

    });

  };

  // after this the otherState: "some other value" will not be there, so to use that, we can copy it into the

  // setState or we can use another useState({otherState: 'some other value'})

  return (

    <div className="App">

      <h1>Hi, I'm a react app./</h1>

      {/\* onClick={this.switchNameHandler()} => if put paranthesis it will immediately cal, we nly pas refernce \*/}

      <button onClick={switchNameHandler}>Switch Name</button>

      <Person

        name={personsState.persons[0].name}

        age={personsState.persons[0].age}

      />

      <Person

        name={personsState.persons[1].name}

        age={personsState.persons[1].age}

      >

        My hobby: Car racing

      </Person>

      <Person

        name={personsState.persons[2].name}

        age={personsState.persons[2].age}

      />

    </div>

  );

};

export default app;

45. stateless vs statefull:

stateless are presentational components and use it more often.

Easier to maintain and manage

Stateful:

Which contains state like useState()

46. PASSING METHOD REFERENCES BETWEEN COMPONENTS:

switchNameHandler = () => {

    // console.log("Was clicked!!!");

    // DONT DO THIS: this.state.persons[0].name = 'sanjay kumar';

    this.setState({

      persons: [

        { name: "sanjay kumar", age: "23" },

        { name: "bujji", age: "22" },

        { name: "pintu", age: "16" },

      ],

    });

  };

 <Person

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

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

          click={this.switchNameHandler}

        >

We can pass the method reference like the above.

* If we want to send the data to method, we can do like below,
* switchNameHandler = (newName) => {
* // console.log("Was clicked!!!");
* // DONT DO THIS: this.state.persons[0].name = 'sanjay kumar';
* this.setState({
* persons: [
* { name: newName, age: "23" },
* { name: "bujji", age: "22" },
* { name: "pintu", age: "16" },
* ],
* });
* };

<button onClick={() => this.switchNameHandler("Sanjay kumar!!")}>

          Switch Name

        </button>

 <button onClick={this.switchNameHandler.bind(this, 'sanjay kumar')}>Switch Name</button>

        <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}

          click={this.switchNameHandler.bind(this, "sanjay!!!")}

        >

          My hobby: Car racing

        </Person>

Here bind() is used to bind the new values into the method. Bind(this, “sanjay”) -> “this” will refer to the method that is calling. And “sanjay” is the value that is passed inside the method.

We can also have arrow func instead of bind method also. But this inefficient. Mostly use bind() method only.

47. ADDING TWO WAY BINDING:

nameChangedHandler = (event) => {

    this.setState({

      persons: [

        { name: "sanjay", age: "23" },

        { name: event.target.value, age: "22" },

        { name: "pintu", age: "15" },

      ],

    });

  };

<Person

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

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

          click={this.switchNameHandler.bind(this, "sanjay!!!")}

          changed={this.nameChangedHandler}

        >

const person = (props) => {

  return (

    <div>

      <p onClick={props.click}>Hi, Im {props.name}</p>

      <p>age: {props.age}</p>

      <p>{props.children}</p>{" "}

      {/\* children special keyword - it puts content between the tags \*/}

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

    </div>

  );

};

48. Adding styling with stylesheets:

Person.css

.Person {

    width: 60%;

    margin: 16px auto;

    border: 1px solid #eee;

    box-shadow: 0 2px 3px #ccc;

    padding: 16px;

    text-align: center;

}

And can import it in js only because of webpack(like a predefined lib in react-js).

Person.js

import React from 'react';

import './Person.css';

const person = ( props ) => {

    return (

        <div className="Person">

            <p onClick={props.click}>I'm {props.name} and I am {props.age} years old!</p>

            <p>{props.children}</p>

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

        </div>

    )

};

export default person;

49. Working with inline styling:

Using hover effect in react inline is pretty difficult and its like a limitation.

// inside class component, render() method

render () {

    const style = {

      backgroundColor: 'white',

      font: 'inherit',

      border: '1px solid blue',

      padding: '8px',

      cursor: 'pointer'

    };

    return (

      <div className="App">

        <h1>Hi, I'm a React App</h1>

        <p>This is really working!</p>

        <button style={style} onClick={() => this.switchNameHandler('Maximilian!!')}>Switch Name</button>

**51. useful resouces and links:**

* create-react-app: <https://github.com/facebookincubator/create-react-app>
* Introducing JSX: <https://reactjs.org/docs/introducing-jsx.html>
* Rendering Elements: <https://reactjs.org/docs/rendering-elements.html>
* Components & Props: <https://reactjs.org/docs/components-and-props.html>
* Listenable Events: <https://reactjs.org/docs/events.html>

**SECTION 4 : WORKING WITH LIST AND CONDITIONALS:**

**(i) Rendering content conditionally:**

To render it conditionally, we should include the parts which should be render conditionally into a single div.

Then enclose that totally it into a curly brace, so now we can use javascript content here.

But “if” statement will not work if it is a block of statements.

But we can use ternary operator to make this work.

Ternary operator: showPerson ? <div> : null

state = {

    persons: [

      { name: "sanjay", age: "23" },

      { name: "bujji", age: "22" },

      { name: "pintu", age: "15" },

    ],

    otherState: "some other value",

    showPersons: false,

  };

togglePersonsHandler = () => {

    const doesShow = this.state.showPersons;

    this.setState({ showPersons: !doesShow });

  };

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

          Toggle Persons

       </button>

{this.state.showPersons === true ? (

          <div>

            <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}

              click={this.switchNameHandler.bind(this, "sanjay!!!")}

              changed={this.nameChangedHandler}

            >

              My hobby: Car racing

            </Person>

            <Person

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

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

            />

          </div>

        ) : null}

**(ii) Handling dynamic content “the javascript way”**

If code goes longer and if we need more ternary operator it will be a mess. Will be confusing. The best way to do:

When react renders or re-renders everything within the render() method will be rendered into the screen.

And we can take advantage of this, and we can use javascript expressions. Above the jsx we can use any javascript code.

Preferred way:

Creating if statement outside the return and storing jsx inside the variable.

And later rendering it by just calling {variableName} in the return statement.

render() {

    const style = {

      backgroundColor: "white",

      font: "inherit",

      border: "1px solid blue",

      padding: "8px",

      cursor: "pointer",

    };

    let persons = null;

    if (this.state.showPersons) {

      persons = (

        <div>

          <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}

            click={this.switchNameHandler.bind(this, "sanjay!!!")}

            changed={this.nameChangedHandler}

          >

            My hobby: Car racing

          </Person>

          <Person

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

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

          />

        </div>

      );

    }

    return (

      <div className="App">

        <h1>Hi, I'm a react app./</h1>

        {/\* onClick={this.switchNameHandler()} => if put paranthesis it will immediately cal, we nly pas refernce \*/}

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

          Toggle Persons

        </button>

        {persons}

      </div>

    );

  }

}

**(iii) Outputting Lists:**

Convert an array of javascript into jsx to valid jsx. So here we use map() function.

{this.state.persons.map((person) => {

            return <Person name={person.name} age={person.age} />;

          })}

No need of repeating Person component many times. Instead we use map or list function to render everything inside the list.

**(iv) Lists and State:**

deletePersonHandler = (personIndex) => {

    const persons = this.state.persons;

    persons.splice(personIndex, 1);

    this.setState({ persons: persons });

  };

let persons = null;

    if (this.state.showPersons) {

      persons = (

        <div>

          {this.state.persons.map((person, index) => {

            return (

              <Person

                name={person.name}

                age={person.age}

                click={() => this.deletePersonHandler(index)} // alternative is bind(, "index")

              />

            );

          })}

        </div>

      );

    }

Note:

If we want to pass any parameter, then

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

if there is no parameter then, this will work.

click={this.deletePersonHandler}

**(v) updating state immutably:**

the above method is not preferable, so instead getting the pointer of the list,

i) we should have a copy of that list. So to have that copy we can use slice method.

const persons = this.state.persons.slice();

    persons.splice(personIndex, 1);

    this.setState({ persons: persons });

ii) or we can use a spread operator to have a copy of that list

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

    persons.splice(personIndex, 1);

    this.setState({ persons: persons });

update the state in a immutable fashion. For this we should have a copy.

**(vi) Lists & Keys:**

When rendering list of data, we should have a key property.

Key property helps react update the list efficiently.

state = {

    persons: [

      { id: "1", name: "sanjay", age: "23" },

      { id: "2", name: "bujji", age: "22" },

      { id: "3", name: "pintu", age: "15" },

    ],

    otherState: "some other value",

    showPersons: false,

  };

<Person

                name={person.name}

                age={person.age}

                click={() => this.deletePersonHandler(index)} // alternative is bind(, "index")

                key={person.id} // unique element

              />

To overcome an error and bring efficiency.

**(vii) Flexible Lists:**

if (this.state.showPersons) {

      persons = (

        <div>

          {this.state.persons.map((person, index) => {

            return (

              <Person

                name={person.name}

                age={person.age}

                click={() => this.deletePersonHandler(index)} // alternative is bind(, "index")

                key={person.id} // unique element

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

              />

            );

          })}

        </div>

      );

    }

**(viii) useful resources & links:**

* Conditional Rendering: <https://reactjs.org/docs/conditional-rendering.html>
* Lists & Keys: <https://reactjs.org/docs/lists-and-keys.html>

**SECTION 5: STYLING REACT COMPONENTS & ELEMENTS:**

**(i) Outlining the problem set:**

* No hover in button and it is inline styled, if we style this button using {button:hover} in App.css this will affect other components too.
* And no dynamic style change.

**(ii) setting styles dynamically:**

Here condition is true, then we can change the style.background to red, which is dynamic in nature.

if (this.state.showPersons) {

      style.backgroundColor = "red";

      persons = (

        <div>

          {this.state.persons.map((person, index) => {

            return (

              <Person

                name={person.name}

                age={person.age}

                click={() => this.deletePersonHandler(index)} // alternative is bind(, "index")

                key={person.id} // unique element

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

              />

            );

          })}

        </div>

      );

    }

**(iii) Setting class names dynamically:**

one way of doing this is creating an array of strings. And joining it to make it as string.

// dynamically adding classes with join to make it as string

    const classes = [];

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

      classes.push("red"); // classes = ["red"]

    }

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

      classes.push("bold"); // classes = ["red", "bold"]

    }

Here we use join to make the array as string.

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

**(iv) Adding and using Radium:**

We cant use {button:hover} in App.css because it will affect every button in other components too.

Its not possible to add “pseudo selector and media query” in javascript variables and make it dynamic.

To do that we have a third party package to add these functionalities to our project.

**“npm install --save radium”**

* Radium is a popular package for react, which allows inline styles with pseudo selectors and media query.

**How to use it?**

import Radium from "radium";

export default Radium(App);

const style = {

      backgroundColor: "green",

      color: "white",

      font: "inherit",

      border: "1px solid blue",

      padding: "8px",

      cursor: "pointer",

**":hover": {**

**backgroundColor: "lightgreen",**

**color: "black",**

**},**

    };

  style.backgroundColor = "red";

**style[":hover"] = {**

**backgroundColor: "salmon",**

**color: "black",**

**};**

**(v) Using Radium for media queries:**

For media query & key frames to work, we need to wrap the main App component with <StyleRoot>. Else we will have a error.

**Person.js**

const style = {

    "@media (min-width: 500px)": {

      width: "450px",

    },

  };

**App.js**

import Radium, { StyleRoot } from "radium";

And return in App.js should be wrapped,

return (

      <StyleRoot>

        <div className="App">

          <h1>Hi, I'm a react app./</h1>

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

          {/\* onClick={this.switchNameHandler()} => if put paranthesis it will immediately cal, we nly pas refernce \*/}

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

            Toggle Persons

          </button>

          {persons}

        </div>

      </StyleRoot>

    );

**(vi) Introducing Styled Components(other styling package):**

Styled Components is a library to make styling very simple.

**styled-components.com**

Install:

**“npm install --save styled-components”**

We can remove radium now because we going to use styled-components now.

import styled from "styled-components";

// styled.div is already a react component. `` its like ()

const StyleDiv = styled.div`

  width: 60%;

  margin: 16px auto;

  border: 1px solid #eee;

  box-shadow: 0 2px 3px #ccc;

  padding: 16px;

  text-align: center;

  @media (min-width: 500px) {

    width: 450px;

  }

`;

return (

    //<div className="Person" style={style}>

    <StyleDiv>

      <p onClick={props.click}>Hi, Im {props.name}</p>

      <p>age: {props.age}</p>

      <p>{props.children}</p>

      {/\* children special keyword - it puts content between the tags \*/}

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

      {/\* two way binding \*/}

    </StyleDiv>

  );

StyleDiv is the style component what we created.

And these are put in head tag as a file. So we can use normal css syntax.

**(vii) More on styled components:**

Creating components for this will be in caps.

const StyledButton = styled.button`

  background-color: green;

  color: white;

  font: inherit;

  border: 1px solid blue;

  padding: 8px;

  cursor: pointer;

  &:hover { ${‘’ /\* & is used before pseudo selectors. \*/}

    background-color: lightgreen;

    color: black;

  }

`;

return (

      <div className="App">

        <h1>Hi, I'm a react app./</h1>

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

        {/\* onClick={this.switchNameHandler()} => if put paranthesis it will immediately cal, we nly pas refernce \*/}

        <StyledButton onClick={this.togglePersonsHandler}>

          Toggle Persons

        </StyledButton>

        {persons}

      </div>

    );

**(viii) Styled Components & Dynamic Styles:**

to dynamically insert into the backticks `` in styled components, its easy, we can use **“background-color: ${props => props.alt ? ‘red’ : ‘green’}”**

const StyledButton = styled.button`

  background-color: ${(props) => (props.alt ? "red" : "green")};

  color: white;

  font: inherit;

  border: 1px solid blue;

  padding: 8px;

  cursor: pointer;

  &:hover {

    background-color: ${(props) => (props.alt ? "salmon" : "lightgreen")};

    color: black;

  }

`;

return (

      <div className="App">

        <h1>Hi, I'm a react app./</h1>

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

        {/\* onClick={this.switchNameHandler()} => if put paranthesis it will immediately cal, we nly pas refernce \*/}

        <StyledButton

          alt={this.state.showPersons}

          onClick={this.togglePersonsHandler}

        >

          Toggle Persons

        </StyledButton>

        {persons}

      </div>

    );

**(ix) working with css modules:**

Css modules will give scope to the styles, like only where it can be used.

How to activate css modules?

**(1)** If the react-scripts or version 1.0.3 then follow:

npm run eject

then in config folder -> webpack.config.dev.js -> find(test: /\.css$/, -> in this there will be -> options -> add:

modules: true,

localIdentName: ‘[name]\_\_[local]\_\_[hash:base64:5]’

// this localIdent will dynamically generate unique css classes.

And copy this modules and localIdent into “webpack.config.prod.js”

**(2)** If react-script version is more than 1, like 2, 4 etc then follow:

Make css file extension as {name.module.css}

Its normally like importing a component.

import classes from "./App.module.css";

let btnClass = classes.Button;

btnClass = btnClass + " " + classes.Red;

return (

      <div className={classes.App}>

        <h1>Hi, I'm a react app./</h1>

        <p className={assignedClasses.join(" ")}>This is really Working!!!</p>

        {/\* onClick={this.switchNameHandler()} => if put paranthesis it will immediately cal, we nly pas refernce \*/}

        <button className={btnClass} onClick={this.togglePersonsHandler}>

          Toggle Persons

        </button>

        {persons}

      </div>

    );

.Button {

  background-color: green;

  color: white;

  font: inherit;

  border: 1px solid blue;

  padding: 8px;

  cursor: pointer;

}

.Button:hover {

  background-color: lightgreen;

  color: black;

}

.Button.Red {

  background-color: red;

}

.Button.Red:hover {

  background-color: salmon;

  color: black;

}

**ANOTHER APPROCH:**

let btnClass = '';

 btnClass = classes.Red; // inside if condition

**and in css:**

.App button {

  background-color: green;

  color: white;

  font: inherit;

  border: 1px solid blue;

  padding: 8px;

  cursor: pointer;

}

.App button:hover {

  background-color: lightgreen;

  color: black;

}

.App button.Red {

  background-color: red;

}

.App button.Red:hover {

  background-color: salmon;

  color: black;

}

**(x) Css modules and media queries:**

How to use a media query through css modules:

Here media query will work without doing nothing.

Through modules we can keep js and css sin separate file and it gives a plus of having a scoping to only the particular component.

By the way, if you somehow also want to define a global (i.e. un-transformed) CSS class in such a .css  file, you can prefix the selector with :global .

**Example:**

:global .Post { ... }

Now you can use className="Post"  anywhere in your app and receive that styling.

**(xi) useful resources:**

* Using CSS Modules in create-react-app Projects: <https://medium.com/nulogy/how-to-use-css-modules-with-create-react-app-9e44bec2b5c2>
* More information about CSS Modules: <https://github.com/css-modules/css-modules>

**SECTION 6: DEBUGGING REACT APPS:**

**(i) Understanding error messages:**

Check the console to get the error and understand or check the line no. where the error occurs and check on that.

**(ii) Finding logical errors by using dev tools & sorucemaps:**

nameChangedHandler = (event, id) => {

    const personIndex = this.state.persons.findIndex((p) => {

      return p.userId === id;

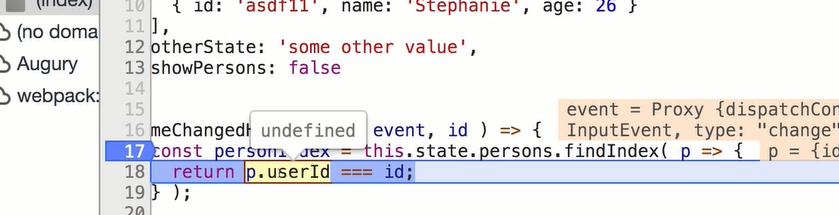
    });

here p.userId is wrong, it should be p.id. Its like undefined === id which is defined.

Then if we start typing, we will not get any error. But we don’t why we cant type.

Solution:

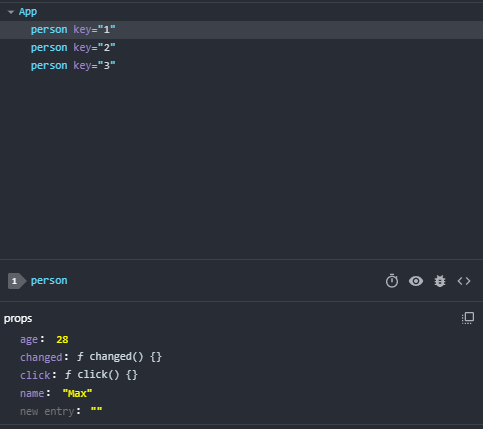
* We can go by line by line in sources
* Then give a breakpoint in the code, where it feels suspicious.
* And using chrome debugger, we can step into line by line and check the output.



* Here we can see, it is undefined and we can change to correct one.

**(iii) working with react developer tools:**

Sometimes we want to know current state of react app. For that we have a extension can react developer tools through that we can debug.



We can also log the component details into console, by clicking the spider type button.

**(iv) Using error boundaries(React 16+):**

Sometimes the code may fail at runtime, we know that. But we cant guarantee when it will be. So for that atleast we should show a custom error msg to the user.

const person = (props) => {

  const rnd = Math.random();

  if (rnd > 0.7) {

    throw new Error("Something went wrong!!!");

  }

  return (

we created a custom error msg. And to solve this, we have a feature called error boundary.

class ErrorBoundary extends Component {

    state = {

        hasError: false,

        errorMessage: ''

    }

    componentDidCatch = (error, info) => {

        this.setState({ hasError: true, errorMessage: error });

    }

    render() {

        if(this.state.hasError) {

            return <h1>{this.state.errorMessage}</h1>

        } else {

            return this.props.children;

        }

    }

}

componentDidCatch() -> this is a method, which receives error and soe potential info.

Whenever the component is wrapped with the error boundary, this throws an error.

this.props.children -> whatever we wrap inside the error boundary

if (this.state.showPersons) {

      persons = (

        <div>

          {this.state.persons.map((person, index) => {

            return (

              <ErrorBoundary key={person.id}>

                <Person

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

                  name={person.name}

                  age={person.age}

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

                />

              </ErrorBoundary>

            );

          })}

        </div>

      );

ErrorBoundary is a higher order component which simply wraps the component. And because of this we should move the key to the error boundary.

Key should be always in the outer element in the map method.

In development mode, we would see the error as same as before. But in production it will show the error boundary message what we kept.

ErrorBoundary should be used only in cases where we cant control or rectify the error.

**(v) useful resources & links:**

* Error Boundaries: <https://reactjs.org/docs/error-boundaries.html>
* Chrome Devtool Debugging: <https://developers.google.com/web/tools/chrome-devtools/javascript/>

**SECTION 7: DIVING DEEPER INTO COMPONENTS & REACT INTERNALS:**

**(i) Better project structure:**

Containers will contain stateful components like state & ui.

Components will have only stateless or presentational.

**(ii) Splitting an app into components:**

if we have one line then we can use like below,

const persons = (props) => (

);

Directly return brackets.

One line -> it will automatically returns, that’s how es6 arrow works

const persons = (props) =>

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

    return (

      <ErrorBoundary key={person.id}>

        <Person

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

          name={person.name}

          age={person.age}

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

        />

      </ErrorBoundary>

    );

  });

And putting upper code of jsx in app to cockpit for better structure.

const cockpit = (props) => {

  const assignedlasses = [];

  let btnClass = "";

  if (props.showPersons) {

    btnClass = classes.Red;

  }

  if (props.personsLength <= 2) {

    assignedlasses.push(classes.red); // classes = ['red']

  }

  if (props.personsLength <= 1) {

    assignedlasses.push(classes.bold); // classes = ['red', 'bold']

  }

  return (

    <div className={classes.Cockpit}>

      <h1>Hi, I'm a React App</h1>

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

      <button className={btnClass} onClick={props.clicked}>

        Toggle Persons

      </button>

    </div>

  );

};

In App.js

 render() {

    let persons = null;

    if (this.state.showPersons) {

      persons = <Persons

            persons={this.state.persons}

            clicked={this.deletePersonHandler}

            changed={this.nameChangedHandler}

          />;

    }

    return (

      <div className={classes.App}>

        <Cockpit

          showPersons={this.state.showPersons}

          personsLength={this.state.persons.length}

          clicked={this.togglePersonsHandler}

        />

        {persons}

      </div>

    );

Now the code is clean. Having separate components is good and efficient.

**(iii) comparing stateless vs stateful:**

Presentational(stateless) vs container(stateful)

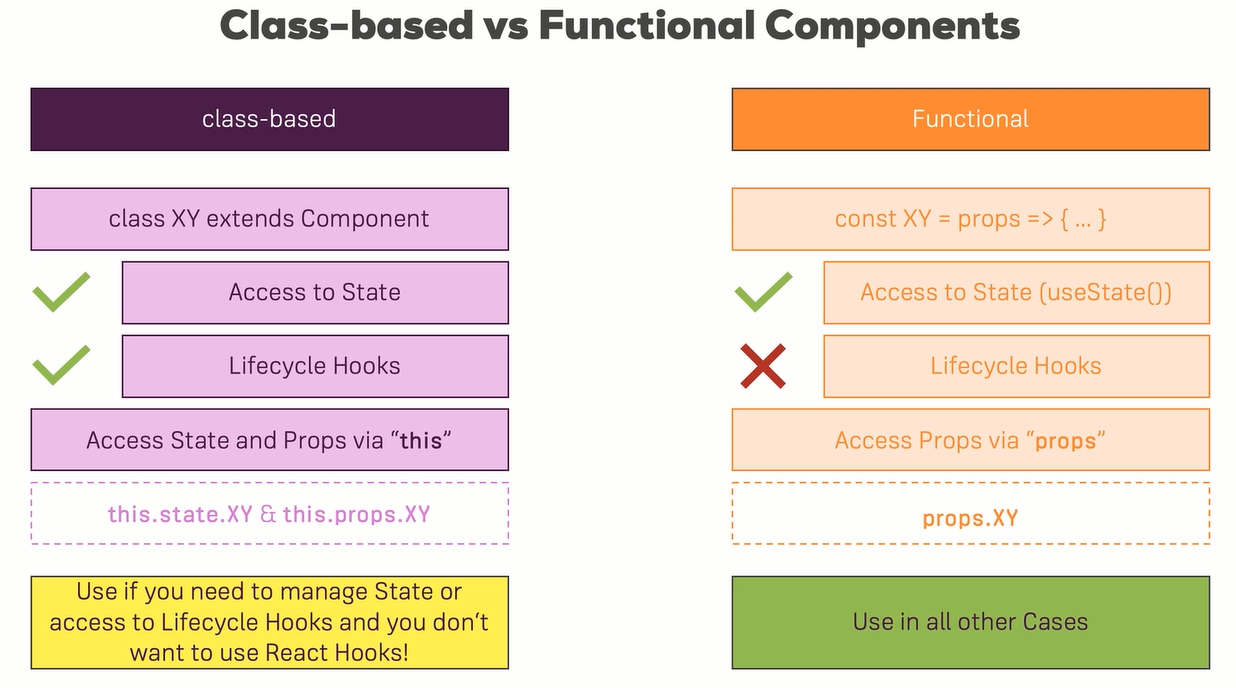
Presentataional:

* Is called like dumb or stateless component. From react 16.8, functional component can use state using react hook, useState.
* Before that only class-based component only can manage state.
* Functional component that doesn’t manage state.
* Majority should be presentational, Why?
* It helps in managing the project. And we know where does the state changes.

Container:

* It manages state and can modify it.
* It is also called stateful.
* Before react 16, only class can have state and functional component cant.
* But after that, even functional component can have.

**(iv) Class-based vs functional components:**



state using in class component:

<Cockpit

          showPersons={this.state.showPersons}

          personsLength={this.state.persons.length}

          clicked={this.togglePersonsHandler}

        />

Props in class component:

Index.js

 <React.StrictMode>

    <App appTitle="Person Manager" />

  </React.StrictMode>,

App.js

<Cockpit

          title={this.props.appTitle}

          showPersons={this.state.showPersons}

          personsLength={this.state.persons.length}

          clicked={this.togglePersonsHandler}

        />

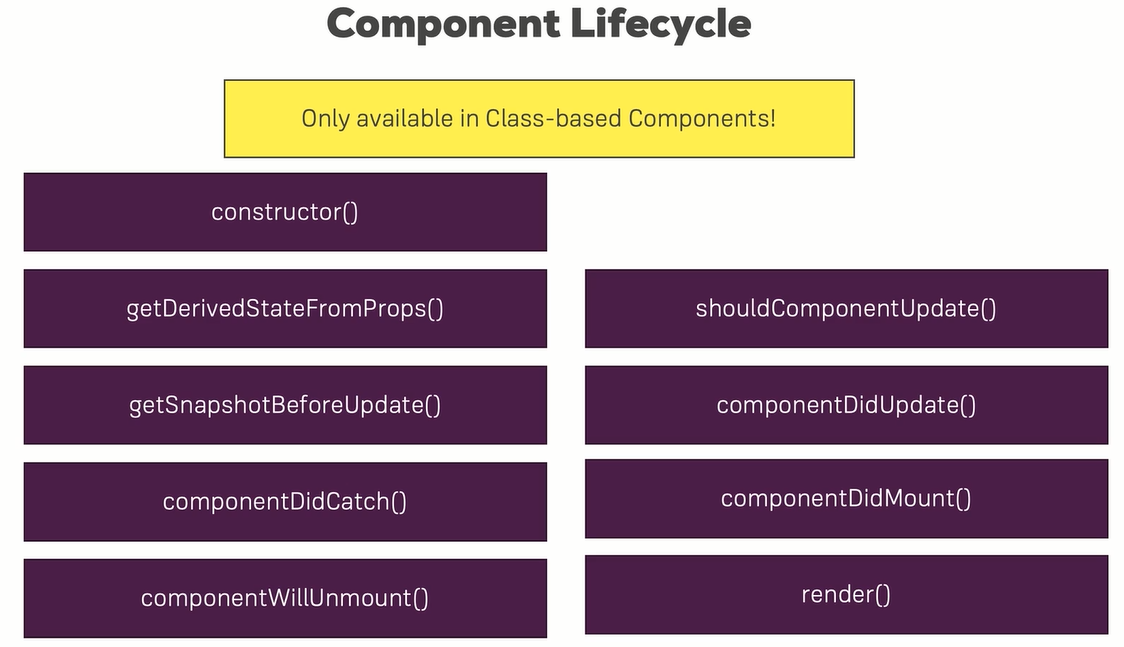
Cockpit.js

<div className={classes.Cockpit}>

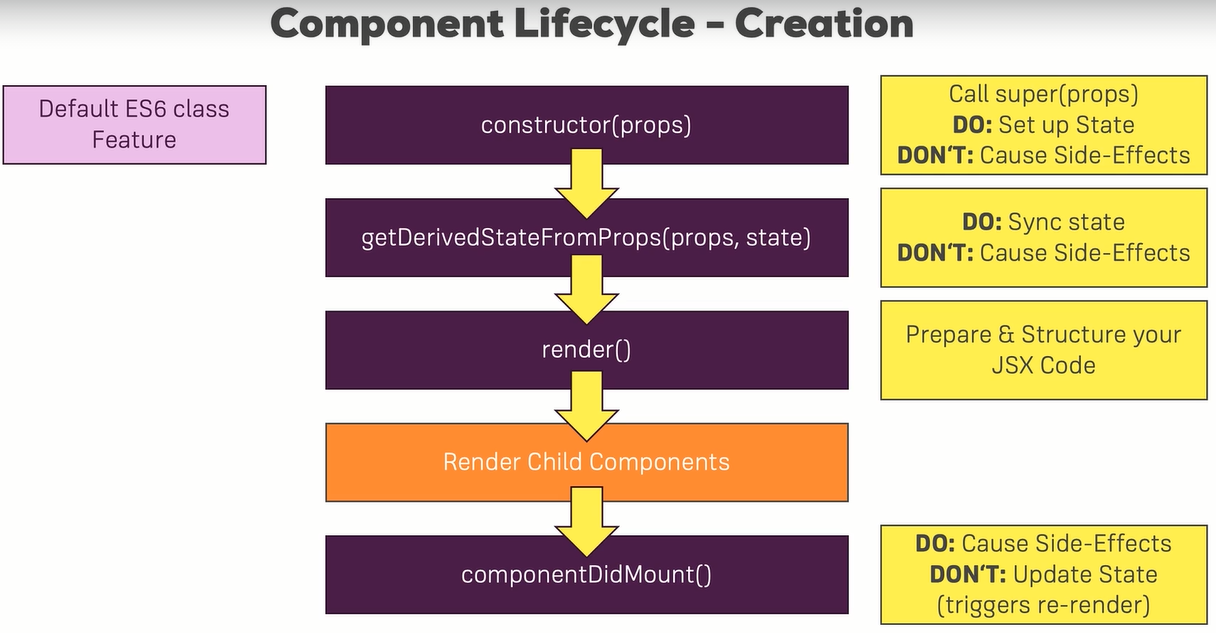
      <h1>{props.title}</h1>

This is how we use state and props in class based component.

**(v) class component lifecycle overview:**



**Component lifecycle - creation:**



constructor(props) {

    super(props);

    console.log('[App.js] constructor');

    // this.state ={}  // setting or initializing the state.

  }

static getDerivedStateFromProps(props, state) {

    console.log("[App.js] getDerivedStateFromProps", props);

    return state;

  }

based on props setting/updating the state.

render() {

    console.log("[App.js]  render");

}

componentDidMount() {

    console.log("[App.js] componentDidMount");

  }

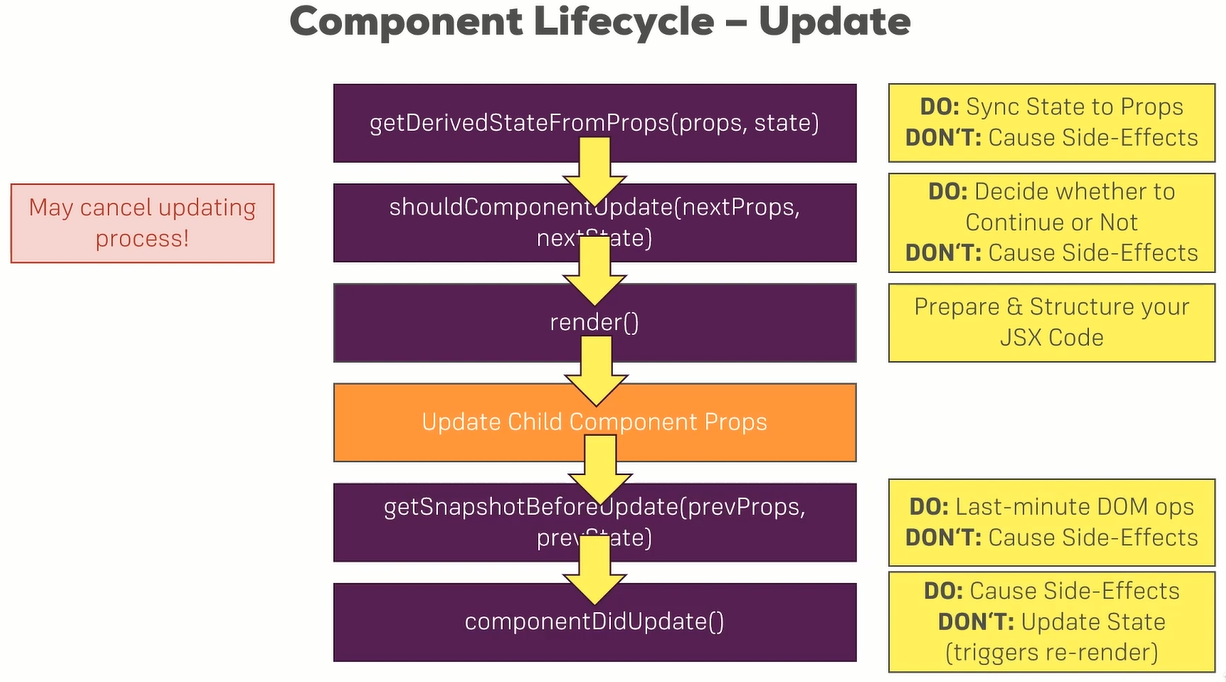
componentWillMount() {

    console.log('[App.js] componentWillMount')

  }

componentWillMount is deprecated!!!

**(vi) Component update lifecycle (for props changes):**



 shouldComponentUpdate(nextProps, nextState) {

    console.log("[Persons.js] shouldComponentUpdate");

    return true;

  }

  getSnapshotBeforeUpdate(prevProps, prevState) {

    console.log("[Persons.js] getSnapshotBeforeUpdate");

    return { message: 'Sanpshot!' };

  }

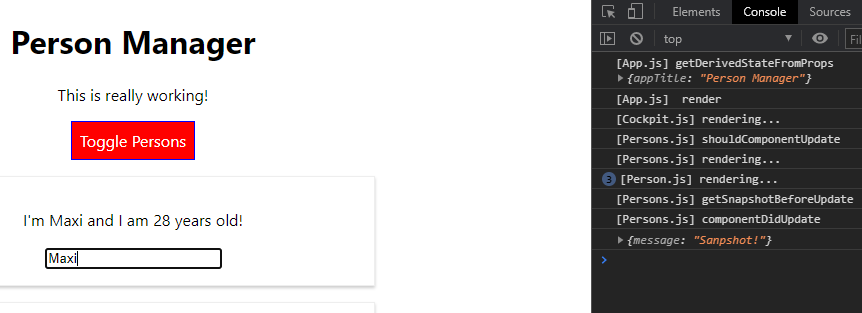
  componentDidUpdate(prevProps, prevState, snapshot) {

    console.log("[Persons.js] componentDidUpdate");

    console.log(snapshot);

  }

After typing a letter in input, which changes the state



getSnapshotUpdate will return some value and that value can be get in the componentDidUpdate(prevProps, prevState, snapshot).

componentWillReceiveProps() {}

this life cycle hook is removed.

**(vii) component update lifecycle (for state changes):**

shouldComponentUpdate(nextProps, nextState) {

    console.log("[App.js] shouldComponentUpdate");

    return true;

  }

  componentDidUpdate() {

    console.log("[App.js] componentDidUpdate");

  }

componentDidMount & componentDidUpdate typically do things like get data from server.

shouldComponentUpdate – used for increasing performance

**(viii) using useEffect() in functional components:**

* We gonna use life cycle hooks in functional components using useEffect.
* useEffect is second most imp, it basically combines all class based life cycle hooks and react hooks.
* We can use it anywhere in functional component body.
* useEffect at default takes a function that runs for every render cycle.

useEffect(() => {});

* So here componentDidMount and componentDidUpdate combine in one effect bcs it runs for every re-render cycle like mount and update do.
* To use **getDerivedStateFromProps()** in functional component, we can use the useState() itself to set the state based on props.

const Cockpit = (props) => {

  useState({

    props // check proper code to set

  });

  useEffect(() => {

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

    // http request can be sent

  });

}

**(ix) controlling the useEffect behaviour:**

This will call the useEffect everytime the component renders.

useEffect(() => {

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

    // http request...

    setTimeout(() => {

      alert("Saved data to cloud");

    }, 1000);

  });

Faking http request here.

Whenever it re-renders but this is not what we need.

**How should I control like, when the persons data is changed then only it should call useEffect; that can be done in shouldComponentUpdate and componentDidUpdate?**

Solution:

useEffect(() => {

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

    // http request...

    setTimeout(() => {

      alert("Saved data to cloud");

    }, 1000);

  }, [props.persons]); // can have multiple fields inside the array

Should update only persons change, so give second argument.

We can use as manay useEffect we need.

By doing like the above, it only calls the useEffect only the persons or name of the person changes.

const Cockpit = (props) => {

  useEffect(() => {

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

    // http request...

    setTimeout(() => {

      alert("Saved data to cloud");

    }, 1000);

  }, [props.showPersons]); // like [a,b,c]

When I gave this props.showPersons, this means when the showPersons data is changed then only it re-renders. Now changing the persons data will not make it to re-render or call the useEffect.

**Now how to change like only it should call useEffect for the first time its mounted?**

**What would we do if we want send http request only once(i.e during mount) and not every render recycle(i.e during update)?**

const Cockpit = (props) => {

  useEffect(() => {

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

    // http request...

    setTimeout(() => {

      alert("Saved data to cloud");

    }, 1000);

  }, []);

If the array is empty, then there is no dependency, if there is no dependency then it will not render again, it will render only once that is the default thing.

**For componentDidMount, we can use an empty array in useEffect.**

**(x) Cleaning up with lifecycle hooks & useEffect():**

- persons.js component will be removed from the DOM, when we click toggle persons.

- for this reason, we will need to cleanup some event listeners you setup somewhere, to do some cleanup work.

- you should have to do some cleanup works

componentWillUnmount() {

      console.log('[Persons.js] componentWillUnmount');

  }

- this will do the cleanup works in class based component.

- for functional component,

const Cockpit = (props) => {

  useEffect(() => {

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

    // http request...

    setTimeout(() => {

      alert("Saved data to cloud");

    }, 1000);

    return () => {

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

    };

  }, []);

- this return stmt will run at last like unmount

- to be more precise, it runs before the main useEffect functions runs, but after the (first) render cycle.

- initially it doesn’t work bcs cockpit is never removed and this acts as componentWillUnmount, only when the component is about to remove this return will run.

- for that we add a button to remove the cockpit.

<div className={classes.App}>

        <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}

      </div>

- when the button is clicked, the component which is already mounted will be unmounted, so only return() in useEffect is called.

useEffect(() => {

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

    // http request...

    setTimeout(() => {

      alert("Saved data to cloud");

    }, 1000);

    return () => {

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

    };

  }, []);

  useEffect(() => {

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

    return () => {

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

    };

  });   // no second argument so not controlling this effect.

- two useEffect is used, one with controlling the useEffect and other one without controlling the useEffect.

**(xi) Cleanup work with useEffect() – Example:**

useEffect(() => {

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

    // http request...

    const timer = setTimeout(() => {

      alert("Saved data to cloud");

    }, 1000);

    return () => {  // this runs when unmounted

      clearTimeout(timer);

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

    };

  }, []);  // this runs only mounted

Before the clearTimeout(), if we remove cockpit, the alert will come after it. But after this, alert will come only once, and if we click remove cockpit, again alert wont come due to clearTimeout() in the return which acts as a componentWillUnmount().

**(xii) using shouldComponentUpdate for optimization:**

 shouldComponentUpdate(nextProps, nextState) {

    console.log("[Persons.js] shouldComponentUpdate");

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

        return true;

    } else {

        return false;

    }

  }

If cockpit is removed, persons which is already mounted will not render because of this shouldComponentUpdate() which increase optimization.

And we use !==, but we are checking the arrays which will be pointing to the references. So it will return false all the time, because the pointer will be always pointing the same thing. But here in our code its not the case, why because,

 nameChangedHandler = (event, id) => {

    const personIndex = this.state.persons.findIndex((p) => {

      return p.id === id;

    });

    const person = {

      ...this.state.persons[personIndex],

    };

    // const person = Object.assign({}, this.state.persons[personIndex]);

    person.name = event.target.value;

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

    persons[personIndex] = person;

    this.setState({ persons: persons });

  };

In the above function, we didn’t directly point to the reference, we create a copy of it and edit it and replace in the state.

So this will always create a new reference. Because of this only, our code runs fine.

Check last min of this video…

**(xiii) Optimizing functional components with React.memo():**

For optimizing we can use shouldComponentUpdate but it is only for class based components.

In functional component for optimization, we can use **React.memo()**, which reacts memorization.

It is technique, where react stores the snapshot of the component and only when this component changes it re-renders else it doesn’t.

export default React.memo(Cockpit);

This alone doesn’t optimize it. Why it doesn’t work?

if (props.persons.length <= 2) {

    assignedlasses.push(classes.red); // classes = ['red']

  }

  if (props.persons.length <= 1) {

    assignedlasses.push(classes.bold); // classes = ['red', 'bold']

  }

It doesn’t work because here the props include persons and then we are getting length from that and using. Using props and dot function makes it to not to work. To make it work, we should pass props as below,

if (props.personsLength <= 2) {

    assignedlasses.push(classes.red); // classes = ['red']

  }

  if (props.personsLength <= 1) {

    assignedlasses.push(classes.bold); // classes = ['red', 'bold']

  }

Now it is working, when typing something in the text bar, there cockpit won’t re-render.

**(xiv) When should u optimize?**

Mostly we do here is, when parent component automatically child component updates, because it depends on parent component.

In shouldComponentUpdate we use code to improvise but it doesn’t come free. If 60% of our project files should update if their parent updates, while if we wrap all this component with the extra check with shouldCompoenntUpdate, then we are running unnecessary checks on 60% of our component code base and that is not we should do.

We should evalute this carefully, if this component is part of parent component, that could change related to something doesn’t affect this at all. Then adding this extra code make no sense and slows down the application tiny bit.

So check where it should update, whether we should use this or not and do it.

**(xv) PureComponents instead of shouldComponentUpdate:**

If we want to put a check for all props, then there is a easier way of writing this component.

shouldComponentUpdate(nextProps, nextState) {

    console.log("[Persons.js] shouldComponentUpdate");

    if (

      nextProps.persons !== this.props.persons ||

      nextProps.changed !== this.props.changed ||

      nextProps.clicked !== this.props.clicked

    ) {

      return true;

    } else {

      return false;

    }

  }

Checking all the props in this component.

If we need to check for all props then we can use PureComponent.

import React, { PureComponent } from "react";

import ErrorBoundary from "../ErrorBoundary/ErrorBoundary";

import Person from "./Person/Person";

// no need of brackets because it is a single line of jsx

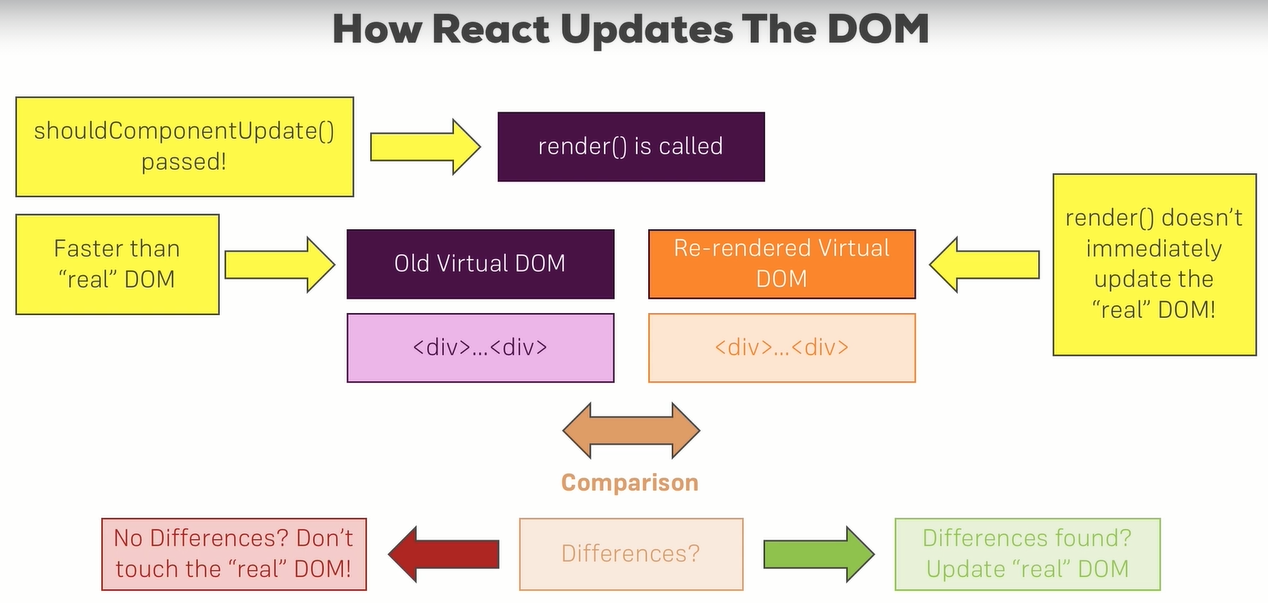
class Persons extends PureComponent {}

PureComponent is a normal component, that already includes shouldComponentUpdate with the complete props check.

Instead of manually checking all props in shouldComponentUpdate, we can use PureComponent.

**(xvi) How react updates the (real) DOM:**

Of course this also applies to functional components and the JSX returned there, NOT just to class-based components with render()!.



Real DOM is touched only when needed.

We will see everything renders using paint flashing in browser console. But what actually happens is when you inspect clicking the toggle button we can see, the div and only the button class is flashed not the cockpit completely it means, it re-renders only the new changes applies to the real DOM else it doesn’t re-renders like the cockpit which doesn’t re-renders which is same.

Here the old virtual DOM is compared with the new virtual DOM and if any changes, then new virtual DOM is updated in the real DOM. If button is the only change then both the DOM’s then only button will be updated in the real DOM.

**(xvii) Rendering adjacent JSX elements:**

Adjacent JSX are not allowed to use separately, it should be enclosed a wrapping div.

class Person extends Component {

  render() {

    console.log("[Person.js] rendering...");

    return [

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

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

        </p>,

        <p>{this.props.children}</p>,

        <input

          type="text"

          onChange={this.props.changed}

          value={this.props.name}

        />

    ];

  }

}

We can wrap it in a array, and use comma after every element to make it work.

But we will have a error, like each child in a list should have unique key. So for that we should give key to each and every elements like below.

class Person extends Component {

  render() {

    console.log("[Person.js] rendering...");

    return [

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

          I'm {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}

          value={this.props.name}

        />

    ];

  }

}

Now everything will be fine only style will not work.

HOC – higher order component, which basically wraps other components.

For that in linux or mac, we can use Aux.js and in windows we can use it as Auxiliary.js because Aux is reserved word in windows.

const aux = props => props.children; // Auxiliary.js

export default aux;

Person.js

import Aux from "../../../hoc/Auxiliary";

<Aux>

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

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

        </p>

        <p>{this.props.children}</p>

        <input

          type="text"

          onChange={this.props.changed}

          value={this.props.name}

        />

      </Aux>

**(xviii) Using React.Fragment:**

In-built Aux is called React.Fragment or Fragment which acts like Aux.

<React.Fragment>

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

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

        </p>

        <p>{this.props.children}</p>

        <input

          type="text"

          onChange={this.props.changed}

          value={this.props.name}

        />

      </React.Fragment>

Or

import React, { Component, Fragment } from "react";

<Fragment>

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

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

        </p>

        <p>{this.props.children}</p>

        <input

          type="text"

          onChange={this.props.changed}

          value={this.props.name}

        />

      </Fragment>

**(xix) Higher Order Components (HOC):**

Aux is a higher order component which doesn’t have any logical codings. Its just a technical wrapper. In Aux we don’t have class in it.

So to have a class, we can create a new js file called WithClass.js which acts as Auxiliary.js but have class in it.

const withClass = props => (

    <div className={props.classes}>

        {props.children}

    </div>

);

App.js

return (

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

      <WithClass classes={classes.App}>

        <button

          onClick={() => {

            this.setState({ showCockpit: false });

          }}

        >

          Remove Cockpit

        </button>

        {this.state.showCockpit ? (

          <Cockpit

            title={this.props.appTitle}

            showPersons={this.state.showPersons}

            personsLength={this.state.persons.length}

            clicked={this.togglePersonsHandler}

          />

        ) : null}

        {persons}

      </WithClass>

      // </div>

    );

**(xx) Another form of HOCs:**

const withClass = (WrappedComponent, className) => {    // normal func

    return props => (                                   // functional component

        <div className={className}>

            <WrappedComponent />

        </div>

    );

};

Here it is a normal func that returns a component.

So in App.js, in import we put it as small letter. And also rename the file name with withClass.js

import withClass from "../hoc/WithClass";

<Aux>

        <button

          onClick={() => {

            this.setState({ showCockpit: false });

          }}

        >

          Remove Cockpit

        </button>

        {this.state.showCockpit ? (

          <Cockpit

            title={this.props.appTitle}

            showPersons={this.state.showPersons}

            personsLength={this.state.persons.length}

            clicked={this.togglePersonsHandler}

          />

        ) : null}

        {persons}

      </Aux>

export default withClass(App, classes.App);

because withClass(wrappedComponent, className).

Which approach we should use?

It depends, there are hoc

If mostly change html code or change some styling and we would say that those best go into your JSX code as a wrapping component, so what we had before, like putting WithClass(component) with class. **(html or styles**)

There are other HOC components that add some behind the scenes logic, some js code that handles error or sends analytics data or anything like that. If logic is imp, then we can use a approach like, (**js logic**)

export default withClass(App, classes.App);

where the withClass acts as a normal function and we concentrate more on logic rather than jsx.

**(xxi) Passing unknown props:**

After making the person.js to make use of withClass hoc, when renders we can find that the name is not printing. Its because the props is not passed into the withClass wrapped component.

To overcome this issue, we should pass the props in the wrapped component in withClass,

const withClass = (WrappedComponent, className) => {    // normal func

    return props => (                                   // functional component

        <div className={className}>

            <WrappedComponent props={props} />

        </div>

    );

};

If we do like this, this will not take all properties it will become a single props, which is invalid. Correct solution is,

const withClass = (WrappedComponent, className) => {    // normal func

    return props => (                                  // functional component

        <div className={className}>

            <WrappedComponent {...props} />

        </div>

); };

Using spread operator in WrappedComponent will copy all the props and make it possible to render the correct props.

**(xxii) Setting State Correctly:**

this.setState({

      persons: persons,

      changeCounter: this.state.changeCounter + 1,

    });

This is the wrong way of setting the state. It doesn’t immediately trigger the update state of component or re-render cycle, it updates only when it has available resources.

this.state doesn’t guarantee the latest state, it may be the latest one.

If changeCounter should be used, then in setState() we can use a function inside.

**BEST PRACTICE:**

this.setState((prevState, props) => {

      return {

        persons: persons,

        changeCounter: prevState.changeCounter + 1,

      };

    });

This is the correct one, which uses prevState.

**(xxiii) Using PropTypes:**

We set state correctly, now we can see how a prop could set.

The user can set the age as any word instead of number. So we can set which prop and which prop type each prop should be. We can do this by adding a package,

npm install --save prop-types

import PropTypes from "prop-types";

// after the component class or functional

Person.propTypes = {

  click: PropTypes.func,

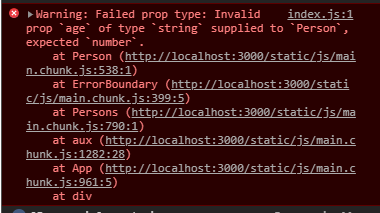
  name: PropTypes.string,

  age: PropTypes.number,

  change: PropTypes.func

};

Below is the error, when we pass the age as string.



If someone uses our package in dev team, this can help in knowing what should they use.

**(xxiv) Using Refs:**

We can use any jsx elements, then we are not limited to setting up two-way binding. To focus that last input what should we do,

class Person extends Component {

  componentDidMount() {

    document.querySelector("input").focus();

  }

}

This will focus only the first and not the last one. Because the querySelector selects the first input. It always works on entire DOM because its not react its from basic browser function.

To focus on particular thing, we can use a concept called, ref. We can use it in any element including the custom element we create.

Ref is just like a key, we can pass a special property.

**Approach 1: (old version)**

class Person extends Component {

  componentDidMount() {

    this.inputElement.focus();

  }

  render() {

    console.log("[Person.js] rendering...");

    return (

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

      <Aux>

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

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

        </p>

        <p>{this.props.children}</p>

        <input

          type="text"

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

          onChange={this.props.changed}

          value={this.props.name}

        />

      </Aux>

      // </div>

    );

Here in ref, we pass a func and create a inputElement dynamically and sets inputEl.

Now it focuses on the last input.

**Approach 2: (new version)**

here we should use constructor & React.createRef().

class Person extends Component {

  constructor(props) {

    super(props);

    this.inputElementRef = React.createRef();

  }

  componentDidMount() {

    // focus only the first input

    // document.querySelector("input").focus();   // bcs querySelector selects the first one.

    // this.inputElement.focus();

    this.inputElementRef.current.focus();

  }

  render() {

    console.log("[Person.js] rendering...");

    return (

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

      <Aux>

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

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

        </p>

        <p>{this.props.children}</p>

        <input

          type="text"

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

          ref={this.inputElementRef}

          onChange={this.props.changed}

          value={this.props.name}

        />

      </Aux>

This above approaches are only for class based component.

**(xxv) Refs with react hooks:**

How to use refs in functional component.

useRef() – not only accessing dom elements but also we can store other elements.

userRef(null) – initial value

const toggleBtnRef = useRef(null);

  toggleBtnRef.current.click();

using click() below itself, will not give opportunity to create a ref for button which is placed below in return().

<button ref={toggleBtnRef} />

To overcome this issue, in class we may run this click() in componentDidMount(). Here we can use useEffect().

const toggleBtnRef = useRef(null);

useEffect(() => {

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

    toggleBtnRef.current.click();

    return () => {

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

    };

  }, []); // this runs only mounted

<button ref={toggleBtnRef} className={btnClass} onClick={props.clicked}>

        Toggle Persons

      </button>

**(xxvi) Understanding prop chain problem:**

state = {

    persons: [

      { id: "1", name: "Max", age: 28 },

      { id: "2", name: "Manu", age: 29 },

      { id: "3", name: "Stephanie", age: 26 },

    ],

    otherState: "some other value",

    showPersons: false,

    showCockpit: true,

    changeCounter: 0,

    authenticated: false,

  };

loginHandler = () => {

    this.setState({ authenticated: true });

  };

 <Cockpit

            title={this.props.appTitle}

            showPersons={this.state.showPersons}

            personsLength={this.state.persons.length}

            clicked={this.togglePersonsHandler}

            login={this.loginHandler}

          />

<Persons

          persons={this.state.persons}

          clicked={this.deletePersonHandler}

          changed={this.nameChangedHandler}

          isAuthenticated={this.state.authenticated}

        />

**Persons.js**

Chaining state of authenticated to persons and persons to person and we gonna use that in person component.

<Person

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

            name={person.name}

            age={person.age}

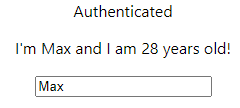
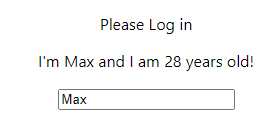
            changed={(event) => this.props.changed(event, person.id)}

            isAuth={this.props.isAuthenticated}

          />

**Person.js**

{this.props.isAuth ? <p>Authenticated</p> : <p>Please Log in</p>}



Props chaining is done here. But the problem is, here we have to pass the props from apps to person. But we will pass it through persons component to send that prop to the person component. It is unnecessary to pass that prop in persons component. This is less reusable.

So to overcome this issue, we can use context api of react to handle cases like this. Do we need data or state in multiple components we no need to send props to other component.

**(xxvii) using the context api:**

Context is like a globally available javascript object. It may not correct, its like, we decide where it should be available.

Technically, it doesn’t have to be an object. We could also have an arrya, string, number etc as a context value.

Without using props we can send objects to other components.

const authContext = React.createContext({

  authenticated: false,

  login: () => {}

});

If we initialize everything in the context to use it in other components, we can get better autocompletion from the IDE that’s the only reason.

AuthContext will be acting like a component. It should wrap all parts of application that needs access to this context. In our demo, the need is in cockpit and persons.

<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}

              login={this.loginHandler}

            />

          ) : null}

          {persons}

        </AuthContext.Provider>

The authContext provider have value part, so only the default initialization in context is not that much imp. The default value will apply only if we don’t set any other value.

In the above code, authenticated and login as it data and this can be passed to appropriate components. Using AuthContext.Consumer.

render() {

    console.log("[Persons.js] rendering...");

    return this.props.persons.map((person, index) => {

      // why return? bcs it not a single line now

      return (

        <ErrorBoundary key={person.id}>

          <Person

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

            name={person.name}

            age={person.age}

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

            isAuth={this.props.isAuthenticated}

          />

        </ErrorBoundary>

      );

    });

  }

render() {

    console.log("[Persons.js] rendering...");

    return (

      <AuthContext.Consumer>

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

          // why return? bcs it not a single line now

          return (

            <ErrorBoundary key={person.id}>

              <Person

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

                name={person.name}

                age={person.age}

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

                isAuth={this.props.isAuthenticated}

              />

            </ErrorBoundary>

          );

        })}

      </AuthContext.Consumer>

    );

  }

AuthContext.Consumer doesn’t take jsx as its child, instead it takes function between the consumer.

So we should make it as a function.

And inside it we will accept context as a parameter to make use of the context api variables which is initialzed in Provider.

But really we should not use it here, because we consume only one isAuth here. So we should put this in Person component.

And isAuth in the above code should not be there move it to person component.

Solution for the abv problem:

App.js

<Aux>

        <button

          onClick={() => {

            this.setState({ showCockpit: false });

          }}

        >

          Remove Cockpit

        </button>

        <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>

      </Aux>

Cockpit.js

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

return (

    <div className={classes.Cockpit}>

      <h1>{props.title}</h1>

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

      <button ref={toggleBtnRef} className={btnClass} onClick={props.clicked}>

        Toggle Persons

      </button>

      <AuthContext.Consumer>

        {(context) => <button onClick={context.login}>Log in</button>}

      </AuthContext.Consumer>

    </div>

  );

Will have a function.

Person.js

<Aux>

        <AuthContext.Consumer>

          {(context) =>

            context.authenticated ? <p>Authenticated</p> : <p>Please Log in</p>

          }

        </AuthContext.Consumer>

It helps in passing props or data directly from A to D.

**(xxviii) contextType & useContext():**

Optimal way to use it in class based component and hooks to use this in functional component.

If we use like before with AuthContext.Consumer then if we need the data in componentDidMount() then there is no chance of using this data there.

But in React 16.6 we can use special property to **contextType** to overcome this.

contextType is recommended to use, instead of AuthContext.Consumer.

// Person.js

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

static contextType = AuthContext;

<Aux>

        {this.context.authenticated ? <p>Authenticated</p> : <p>Please Log in</p>}

We should call it using this.context because it is a react thing which is built-in to use. And this is ony class-based component.

For functional component,

import React, { useEffect, useRef, useContext } from "react";

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

const authContext = useContext(AuthContext);

<button onClick={authContext.login}>Log in</button>

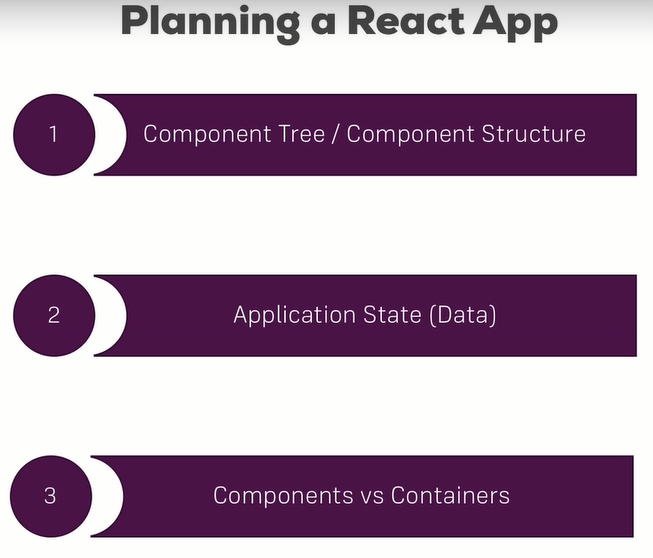
It acts same like the contextType, but this works in functional component.

**(xxix) Useful resources:**

* More on useEffect(): <https://reactjs.org/docs/hooks-effect.html>
* State & Lifecycle: <https://reactjs.org/docs/state-and-lifecycle.html>
* PropTypes: <https://reactjs.org/docs/typechecking-with-proptypes.html>
* Higher Order Components: <https://reactjs.org/docs/higher-order-components.html>
* Refs: <https://reactjs.org/docs/refs-and-the-dom.html>

**SECTION 8: A REAL APP: THE BURGER BUILDER:**

**(i) Planning an App in React:**

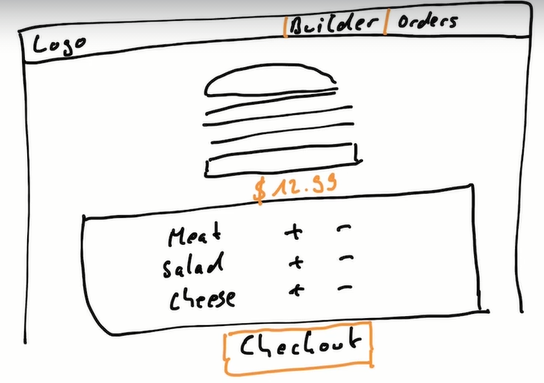


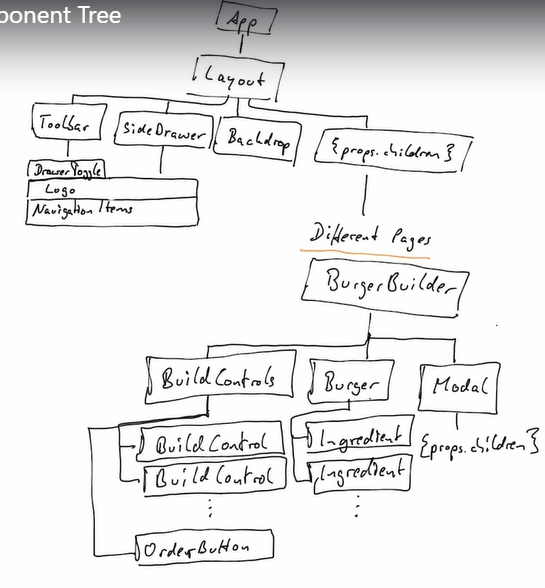
1. what should be the components in component structure in our project. Components count may become higher at end and its fine.

2. what data or state we should use, which we should track.

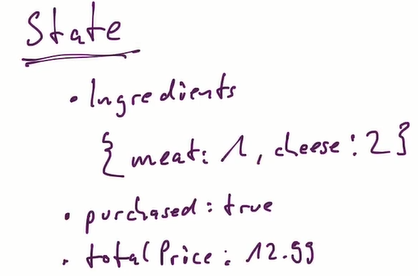
3. and finally we can consider which should be components and which should be containers.

**(ii) Planning our app – layout and component tree:**





**(iii) planning our state:**



This state should not be added to App instead it should be added to BurgerBuilder. So BurgerBuilder should be stateful and others should be dump components or stateless.

**(iv) Setting up project:**

Create new project and clean it. Index.css file have styles that apply globally.

**(v) Creating a Layout Component:**

const layout = ( props ) => (

  <Aux>

    <div>Toolbar, Sidebar, Backdrop</div>

    <main>{props.children}</main>

  </Aux>

);

class App extends Component {

  render() {

    return (

      <div>

        <Layout>

          <p>Test</p>

        </Layout>

      </div>

    );

  }

}

**(vi) Implementation of Burger Builder Container:**

class BurgerBuilder extends Component {

    render() {

        return (

            <Aux>

                <div>Burger</div>

                <div>Build Controls</div>

            </Aux>

        );

    }

}

class App extends Component {

  render() {

    return (

      <div>

        <Layout>

          <BurgerBuilder />

        </Layout>

      </div>

    );

  }

}

**(vii) Adding Dynamic Ingredient Component:**

const burgerIngredient = (props) => {

  let ingredient = null;

  switch (props.type) {

    case "bread-bottom":

      ingredient = <div className={classes.BreadBottom}></div>;

      break;

    case "bread-top":

      ingredient = (

        <div className={classes.BreadTop}>

          <div className={classes.Seeds1}></div>

          <div className={classes.Seeds2}></div>

        </div>

      );

      break;

    case "meat":

      ingredient = <div className={classes.Meat}></div>;

      break;

    case "cheese":

      ingredient = <div className={classes.Cheese}></div>;

      break;

    case "bacon":

      ingredient = <div className={classes.Bacon}></div>;

      break;

    case "salad":

      ingredient = <div className={classes.Salad}></div>;

      break;

    default:

      ingredient = null;

  }

  return ingredient;

};

This is the ingredients which take and clear that.

**(viii) Adding prop type validation:**

to use PropTypes it is easy in class component. so we can make this Class Component.

import PropTypes from "prop-types";

BurgerIngredient.propTypes = {

    type: PropTypes.string.isRequired

};

It should be only string.

**(ix) Starting the burger component:**

const burger = (props) => {

  return (

    <div className={classes.Burger}>

      <BurgerIngredient type="bread-top" />

      <BurgerIngredient type="cheese" />

      <BurgerIngredient type="meat" />

      <BurgerIngredient type="bread-bottom" />

    </div>

  );

};

**(x) Outputting burger ingredients dynamically:**

state = {

    ingredients: {

      salad: 1,

      bacon: 1,

      cheese: 2,

      meat: 2,

    },

  };

<Burger ingredients={this.state.ingredients} />

Convert this object into array. To use the map function. Below is that.

Conversion of object into array.

const transformedIngredients = Object.keys(props.ingredients)

  .map((igKey) => {

    return [...Array(props.ingredients[igKey])].map((\_, i) => {

        return <BurgerIngredient key={igKey + i} type={igKey} />

    });

  });

return (

    <div className={classes.Burger}>

      <BurgerIngredient type="bread-top" />

      {transformedIngredients}

      <BurgerIngredient type="bread-bottom" />

    </div>

  );

Here Object.keys() has the object keys in a array like [salad, cheese, meat, bacon]. And we are mapping this array again. And we should create an array of size igKey to render that much times. Like cheese of size 2, will be rendering two times of map internally.

**(xi) Calculating the ingredient sum dynamically:**

const transformedIngredients = Object.keys(props.ingredients)

  .map((igKey) => {

    return [...Array(props.ingredients[igKey])].map((\_, i) => {

        return <BurgerIngredient key={igKey + i} type={igKey} />

    });

  }).reduce((arr, el) => {

      return arr.concat(el)

  }, []);

This thing reduce it to a single array. This is flattening the array.

let transformedIngredients = Object.keys(props.ingredients)

  .map((igKey) => {

    return [...Array(props.ingredients[igKey])].map((\_, i) => {

        return <BurgerIngredient key={igKey + i} type={igKey} />

    });

  }).reduce((arr, el) => {

      return arr.concat(el)

  }, []);

  if(transformedIngredients.length === 0) {

    transformedIngredients = <p>Please Start Adding Ingredients!</p>

  }

**(xii) Adding the Build Control Component:**

const buildControl = (props) => (

    <div className={classes.BuildControl}>

        <div className={classes.Label}>{props.label}</div>

        <button className={classes.Less}>Less</button>

        <button className={classes.More}>More</button>

    </div>

);

**(xiii) Outputting multiple build controls:**

const buildControls = (props) => (

    <div className={classes.BuildControls}>

        {controls.map(ctrl => (

            <BuildControl key={ctrl.label} label={ctrl.label} />

        ))}

    </div>

);

const buildControl = (props) => (

    <div className={classes.BuildControl}>

        <div className={classes.Label}>{props.label}</div>

        <button className={classes.Less}>Less</button>

        <button className={classes.More}>More</button>

    </div>

);

**(xiv) Connecting state to Build Controls:**

addIngredientHandler = (type) => {

    const oldCount = this.state.ingredients[type];

    const updateCount = oldCount + 1;

    const updatedIngredients = {

      ...this.state.ingredients,

    };

    updatedIngredients[type] = updateCount;

    const priceAddition = INGREDIENT\_PRICES[type];

    const oldPrice = this.state.totalPrice;

    const newPrice = oldPrice + priceAddition;

    this.setState({ totalPrice: newPrice, ingredients: updatedIngredients });

  };

**(xv) Removing ingredients safely:**

removeIngredientHandler = (type) => {

    const oldCount = this.state.ingredients[type];

    if (oldCount <= 0) {

      return;

    }

    const newCount = oldCount - 1;

    const updatedIngredients = {

      ...this.state.ingredients,

    };

    updatedIngredients[type] = newCount;

    const priceSubtraction = INGREDIENT\_PRICES[type];

    const oldPrice = this.state.totalPrice;

    const newPrice = oldPrice - priceSubtraction;

    this.setState({ ingredients: updatedIngredients, totalPrice: newPrice });

  };

 render() {

    const disabledInfo = {

      ...this.state.ingredients,

    };

    for(let key in disabledInfo){

        disabledInfo[key] = disabledInfo[key] <= 0;

    } // {salad: true, cheese: false}

**(xvi) Displaying & Updating the Burger Price:**

price = {this.state.totalPrice}

<div className={classes.BuildControls}>

      <p>Current Price: <strong>{props.price.toFixed(2)}</strong></p>

**(xvii) Adding the order button:**

 updatePurchaseState(ingredients) {

    // const ingredients = {    // this may not be updated since we called quickly in other methods

    //   ...this.state.ingredients,

    // };

    const sum = Object.keys(ingredients)

      .map((igKey) => {

        return ingredients[igKey];

      })

      .reduce((sum, el) => {

        return sum + el;

      }, 0);

    this.setState({ purchasable: sum > 0 });

  }

addIngredientHandler = (type) => {

    this.updatePurchaseState(updatedIngredients);

  };

Both in add and remove methods.

**(xviii) Creating the Order Summary Modal:**

const modal = (props) => (

    <div className={classes.Modal}>

        {props.children}

    </div>

);

const orderSummary = (props) => {

  const ingredientSummary = Object.keys(props.ingredients).map((igKey) => {

    return (

      <li key={igKey}>

        <span style={{ textTransform: "capitalize" }}>{igKey}</span>:{" "}

        {props.ingredients[igKey]}

      </li>

    );

  });

  return (

    <Aux>

      <h3>Your Order</h3>

      <p>A Delicious burger with the following ingredients:</p>

      <ul>{ingredientSummary}</ul>

      <p>Continue to checkout?</p>

    </Aux>

  );

};

**BurgerBuilder.js**

<Aux>

        <Modal>

            <OrderSummary ingredients={this.state.ingredients} />

        </Modal>

        <Burger ingredients={this.state.ingredients} />

        <BuildControls

          ingredientAdded={this.addIngredientHandler}

          ingredientRemoved={this.removeIngredientHandler}

          disabled={disabledInfo}

          price={this.state.totalPrice}

          purchasable={this.state.purchasable}

        />

      </Aux>

**(xix) Showing & hiding the modal (with animation!):**

purchaseHandler() {

    this.setState({purchasing: true});

  }

this will cause an error, why? It is a directly a function and this keyword will not refer the current class.

To make this we should make this function as a property to pass it to jsx.

const modal = (props) => (

  <div

    className={classes.Modal}

    style={{

      transform: props.show ? "translateY(0)" : "translateY(-100vh)",

      opacity: props.show ? "1" : "0",

    }}

  >

    {props.children}

  </div>

);

Opacity hides it and shows it conditionally.

**(xx) Implementing the backdrop component:**

**Backdrop.js**

const backdrop = (props) =>

  props.show ? <div className={classes.Backdrop} onClick={props.clicked}></div> : null;

**Modal.js**

<Aux>

    <Backdrop show={props.show} clicked={props.modalClosed} />

    <div

      className={classes.Modal}

      style={{

        transform: props.show ? "translateY(0)" : "translateY(-100vh)",

        opacity: props.show ? "1" : "0",

      }}

    >

      {props.children}

    </div>

**BurgerBuilder.js**

purchaseCancelHandler = () => {

      this.setState({purchasing: false});

  }

<Aux>

        <Modal show={this.state.purchasing} modalClosed={this.purchaseCancelHandler}>

            <OrderSummary ingredients={this.state.ingredients} />

        </Modal>

        <Burger ingredients={this.state.ingredients} />

**(xxi) Adding a custom button component:**

const button = (props) => (

  <button

    onClick={props.clicked}

    className={[classes.Button, classes[props.btnType]].join(" ")}

  >

    {props.children}

  </button>

);

Creating and adding own css styling because we need different styling for cancel and continue button.

**(xxii) Implementing the button component:**

**OrderSummary.js**

 <Aux>

      <h3>Your Order</h3>

      <p>A Delicious burger with the following ingredients:</p>

      <ul>{ingredientSummary}</ul>

      <p>Continue to checkout?</p>

      <Button clicked={props.purchaseCancelled} btnType="Danger">CANCEL</Button>

      <Button clicked={props.purcharseContinued} btnType="Success">CONTINUE</Button>

    </Aux>

**BurgerBuilder.js**

purchaseCancelHandler = () => {

    this.setState({ purchasing: false });

  };

  purchaseContinueHandler = () => {

    alert("You Continue!");

  };

**(xxiii) Adding the price to the Order Summary:**

<p><strong>Total Price: {props.price}</strong></p>

**(xxiv) Adding a toolbar:**

const toolbar = (props) => (

  <header className={classes.Toolbar}>

    <div>MENU</div>

    <div>LOGO</div>

    <nav>...</nav>

  </header>

);

const layout = (props) => (

  <Aux>

    <Toolbar />

    <main className={classes.Content}>{props.children}</main>

  </Aux>

);

**(xxv) Using LOGO in our application:**

**Logo.js**

import burgerLogo from "../../assets/images/burger-logo.png";

const logo = (props) => (

  <div className={classes.Logo}>

    <img src={burgerLogo} alt="MyBurger" />

  </div>

);

**Toolbar.js**

const toolbar = (props) => (

  <header className={classes.Toolbar}>

    <div>MENU</div>

    <Logo />

    <nav>...</nav>

  </header>

);

**(xxvi) Adding reusable navigation items:**

const navigationItem = (props) => (

    <ul className={classes.NavigationItems}>

        <NavigationItem link="/" active>Burger Builder</NavigationItem>    {/\* for boolean, we can give only active \*/}

        <NavigationItem link="/">Checkout</NavigationItem>

    </ul>

);

const navigationItem = (props) => (

  <li className={classes.NavigationItem}>

    <a href={props.link} className={props.active ? classes.active : null}>

      {props.children}

    </a>

  </li>

);

const toolbar = (props) => (

  <header className={classes.Toolbar}>

    <div>MENU</div>

    <Logo />

    <NavigationItems />

  </header>

);

**(xxvii) Creating a responsive sidedrawer:**

Sidedrawer is only visible for small screens like mobiles.

const sideDrawer = (props) => {

    return (

        <div className={classes.SideDrawer}>

            <Logo />

            <nav>

                <NavigationItems />

            </nav>

        </div>

    );

};

const layout = (props) => (

  <Aux>

    <Toolbar />

    <SideDrawer />

    <main className={classes.Content}>{props.children}</main>

  </Aux>

);

**(xxviii) Working on responsive adjacements:**

Adjusting the heights where they are used or embedded. Like setting height of logo to 100% and modifying it in the toolbar and sidedrawer as we need.

(i) To make that we shall pass a prop and set height in inline style of logo.

const logo = (props) => (

  <div className={classes.Logo} style={{height: props.height}}>

    <img src={burgerLogo} alt="MyBurger" />

  </div>

);

const toolbar = (props) => (

  <header className={classes.Toolbar}>

    <div>MENU</div>

    <Logo height="80%" />

    <NavigationItems />

  </header>

);

const sideDrawer = (props) => {

    return (

        <div className={classes.SideDrawer}>

            <Logo height="11%" />

            <nav>

                <NavigationItems />

            </nav>

        </div>

    );

};

(ii) else we can wrap the logo with the div and give inline style to that div and give any props to logo component.

const logo = (props) => (

  <div className={classes.Logo}>

    <img src={burgerLogo} alt="MyBurger" />

  </div>

);

const toolbar = (props) => (

  <header className={classes.Toolbar}>

    <div>MENU</div>

    <div className={classes.Logo}>

      <Logo />

    </div>

    <NavigationItems />

  </header>

);

.Logo {

    height: 80%;

}

const sideDrawer = (props) => {

  return (

    <div className={classes.SideDrawer}>

      <div className={classes.Logo}>

        <Logo />

      </div>

      <nav>

        <NavigationItems />

      </nav>

    </div>

  );

};

.Logo {

    height: 11%;

}

**(xxix) More about responsive adjustments:**

Here we adjust navigation items.

Already defined property in base class, don’t need to put that in media query. Like box-sizing is already there in base class, so need to put that in media for desktop, because this takes box-sizing from base class.

Toolbar.module.css

/\* mobile device - if max width is 499px and minimum can be anything \*/

@media (max-width: 499px) {

    .DesktopOnly {

        display: none;

    }

}

SideDrawer.module.css

@media (min-width: 500px) {    /\* if screen exceeds 500px \*/

    .SideDrawer {

        display: none;

    }

}

**(xxx) Reusing the backdrop:**

class Layout extends Component {

  state = {

    showSideDrawer: true,

  };

  sideDrawerClosedHandler = () => {

    this.setState({ showSideDrawer: false });

  };

  render() {

    return (

      <Aux>

        <Toolbar />

        <SideDrawer

          open={this.state.showSideDrawer}

          closed={this.sideDrawerClosedHandler}

        />

        <main className={classes.Content}>{this.props.children}</main>

      </Aux>

    );

  }

}

Newly added this to layout.

const sideDrawer = (props) => {

    let attachedClasses = [classes.SideDrawer, classes.Close];

    if(props.open) {

        attachedClasses = [classes.SideDrawer, classes.Open];

    }

  return (

    <Aux>

      <BackDrop show={props.open} clicked={props.closed} />

      <div className={attachedClasses.join(' ')}>

        <div className={classes.Logo}>

          <Logo />

        </div>

        <nav>

          <NavigationItems />

        </nav>

      </div>

    </Aux>

  );

};

attachedClasses to open and close the sidedraw using css stylings.

**(xxxi) Adding sidedrawer toggle button:**

**Layout.js**

sideDrawerToggleHandler = () => {

    this.setState((prevState) => {

      return { showSideDrawer: !prevState.showSideDrawer };

    });

  };

<Aux>

        <Toolbar drawerToggleClicked={this.sideDrawerToggleHandler} />

        <SideDrawer

          open={this.state.showSideDrawer}

          closed={this.sideDrawerClosedHandler}

        />

        <main className={classes.Content}>{this.props.children}</main>

      </Aux>

**Toolbar.js**

const toolbar = (props) => (

  <header className={classes.Toolbar}>

    <DrawerToggle clicked={props.drawerToggleClicked} />

    <div className={classes.Logo}>

      <Logo />

    </div>

    <nav className={classes.DesktopOnly}>

      <NavigationItems />

    </nav>

  </header>

);

**DrawerToggle.js**

const drawerToggle = (props) => (

    <div onClick={props.clicked}>MENU</div>

);

Created a separate component drawerToogle.

**(xxxii) Adding a hamburger icon:**

const drawerToggle = (props) => (

    <div className={classes.DrawerToggle} onClick={props.clicked}>

        <div></div>

        <div></div>

        <div></div>

    </div>

);

.DrawerToggle {

  width: 40px;

  height: 100%;

  display: flex;

  flex-flow: column;

  justify-content: space-around;

  align-items: center;

  padding: 10px 0;

  box-sizing: border-box;

  cursor: pointer;

}

.DrawerToggle div {

  width: 90%;

  height: 3px;

  background-color: white;

}

@media (min-width: 500px) {

  .DrawerToggle {

    display: none;

  }

}

{\* this css makes that 3 lines \*}

**(xxiii) Improving the app – introduction:**

Prop types

Life cycles to improve.

**(xxxiv) prop type validation:**

Only in burger ingredients. And not anywhere else.

Because this project is for other people, dev team, api dev etc. Additionally we get logical errors if we assign wrong keys.

**(xxxv) Improving performance:**

We should find where we can improve performance or reduce unnecessary re-rendering.

Like state which changes made the component to re-render.

Even though we are not showing it order summary is rendered.

class OrderSummary extends Component {

  componentDidUpdate() {

    console.log("[OrderSumary.js] DidUpdate");

  }



class Modal extends Component {

  shouldComponentUpdate(nextProps, nextState) {

    return nextProps.show !== this.props.show;

  }

  componentDidUpdate() {

    console.log('[Modal.js] didUpdate')

  }

  render() {

Modal is the wrapping element which also controls the OrderSummary which is a wrapped element.

In Modal PureComponent is not used, because it checks more than we need to check.

**(xxxvi) Using component lifecycle methods:**

Why we didn’t use any life cycle methods, because no use till now, may be after http request we can add.

**(xxxvii) changing the folder structure:**

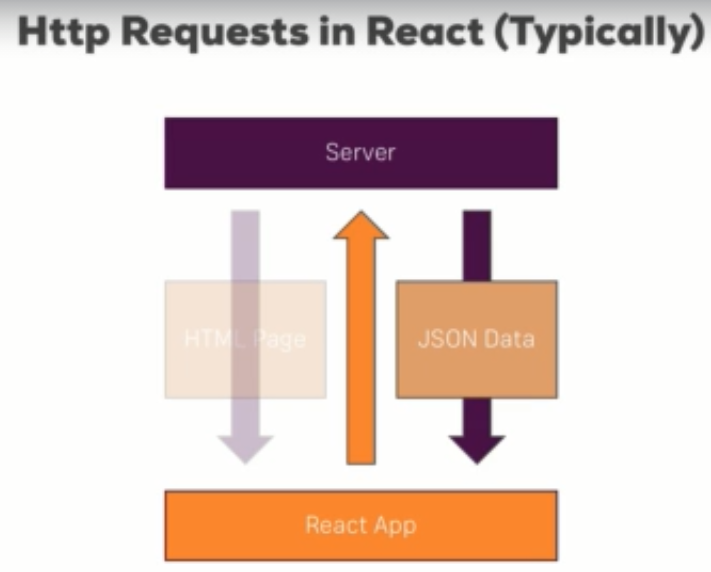
We can put layout into container as it as state or we can put layout into hoc because it wraps the other components and acts as higher order component.

**SECTION 9: REACHING OUT TO THE WEB:**

**(i) Module Introduction:**

Using http/ajax requests from react frontend to the server side.

**(ii) Understanding https request in React:**



Instead of sending html page back from the server we can get a json data from the server and we can send json resource to create something in server database.

Typically, server is a restful API.

**(iii) Understanding our project and introducing axios:**

To create a server like thing we will use,

jsonplaceholder.typeicode.com – which is restful api backend where we can send dummy data to get or even sent data to post.

1. Through XMLHttpRequest, we can create ajax request and send them to speicifc url.
2. Axios – third party library to used to send ajax requests.

**npm install axios –save**

[**https://github.com/axios/axios**](https://github.com/axios/axios) **- tutorial for axios**

**(iv) creating a http request to get data:**

http request is a side effect and componentDidMount is where we can cause side effects and not for updating state.

const posts = axios.get('https://jsonplaceholder.typicode.com/posts');

storing it in const won’t work, because axios sends async request to server. But js executes in a sync manner like line after line it executes.

When it comes to axios this may be late. And js wont wait and will go to next line and that line when we access the const variable of axios will be null or undefined.

Axios uses promises to overcome this issue.

axios.get('https://jsonplaceholder.typicode.com/posts')

        .then();

It uses then() which is a method takes func as input, and it executes once the promise resolves. Then() then receives a response.

componentDidMount() {

    const posts = axios

      .get("https://jsonplaceholder.typicode.com/posts")

      .then((respone) => {

        console.log(respone);

      });

  }

**(v) Rendering fetched data to screen:**

 state = {

    posts: [],

  };

  componentDidMount() {

    const posts = axios

      .get("https://jsonplaceholder.typicode.com/posts")

      .then((respone) => {

        this.setState({ posts: respone.data });

        // console.log(respone);

      });

  }

  render() {

    const posts = this.state.posts.map(post => {

            return <Post key={post.id} title={post.title} />

        }

    );

return (

      <div>

        <section className="Posts">

          {posts}

        </section>

Setting the array of blog data to state. And rendering it again.

**(vi) Transforming Data:**

componentDidMount() {

    console.log("[Blog.js] componentDidMount");

    axios.get("https://jsonplaceholder.typicode.com/posts").then((response) => {

      const posts = response.data.slice(0, 4);

      const updatedPosts = posts.map((post) => {

        return {

          ...post,

          author: "Sanjay",

        };

      });

      this.setState({ posts: updatedPosts });

      // console.log(respone);

    });

  }

Data is shortened using slice() because the response data is array. And updated the post with the author. Like this we can transform data in componentDidMount().

**(vii) Making a post selectable:**

const posts = this.state.posts.map((post) => {

      return (

        <Post

          key={post.id}

          title={post.title}

          author={post.author}

          clicked={() => this.postSelectedHandler(post.id)}

        />

      );

    });

postSelectedHandler = (id) => {

    this.setState({ selectedPostId: id, selectedPost: postSelected });

  };

**(viii) fetching data on update (without creating infinte loops):**

Here we componentDidUpdate and it frequently sends requests because its updating each time a state and it goes continously. And creates a infinite loop.

state = {

    loadedPost: null,

  };

  componentDidUpdate() {

    if (this.props.id) {

      if (

        !this.state.loadedPost ||

        (this.state.loadedPost && this.state.loadedPost.id !== this.props.id)

      ) {

        axios

          .get("https://jsonplaceholder.typicode.com/posts/" + this.props.id)

          .then((response) => {

            // console.log(response);

            this.setState({ loadedPost: response.data });

          });

      }

    }

  }

Here we give some conditions to restrict from infinite loop requests.

**(ix) posting data to server:**

In post request we have to send data to the server. And the data should be a javascript object to be sent to the server. Javascript object is a kind of json data.

Axios will make it stringify to json.

state = {

    title: "",

    content: "",

    author: "Max",

  };

  postDataHandler = () => {

    const post = {

      title: this.state.title,

      body: this.state.content,

      author: this.state.author,

    };

    axios

      .post("https://jsonplaceholder.typicode.com/posts", post)

      .then((response) => console.log(response));

  };

**(x) Sending a DELETE request:**

deletePostHandler = () => {

    axios

      .delete("https://jsonplaceholder.typicode.com/posts/" + this.props.id)

      .then((response) => console.log(response));

  };

**(xi) Handling errors locally:**

componentDidMount() {

    console.log("[Blog.js] componentDidMount");

    axios

      .get("https://jsonplaceholder.typicode.com/postssss")

      .then((response) => {

        const posts = response.data.slice(0, 4);

        const updatedPosts = posts.map((post) => {

          return {

            ...post,

            author: "Sanjay",

          };

        });

        this.setState({ posts: updatedPosts });

        // console.log(respone);

      })

      .catch((error) => {

        // console.log(error);

        this.setState({ error: true });

      });

  }

Made a wrong url, get request and this sends an error. So we can catch that error through axios and can do display some other in UI.

state = {

    posts: [],

    selectedPostId: null,

    error: false,

  };

 let posts = <p style={{ textAlign: "center" }}>Something went wrong!</p>;

    if (!this.state.error) {

      posts = this.state.posts.map((post) => {

        return (

          <Post

            key={post.id}

            title={post.title}

If error is false, then it will print all posts else it will print something went wrong in UI.

**(xii) Adding interceptors to execute code globally:**

[**https://medium.com/swlh/handling-access-and-refresh-tokens-using-axios-interceptors-3970b601a5da**](https://medium.com/swlh/handling-access-and-refresh-tokens-using-axios-interceptors-3970b601a5da)

If some request sent from any component we can execute a code like globally whatever request sent or response return, this code will run.

Interceptors are functions defined globally which will be executed for every request leaving ur app and every response returning into it.

It may useful in authorization, log response or want handle error globally.

All axios import, all share the same configuration.

And I will define this interceptors in index.js to make it globally available for all components because this index.js is the root component.

This interceptor have attribute which is config or request.

In interceptors, we must always return the request or else it will be blocked and no other request is sent.

We can also edit the request config before we return it.

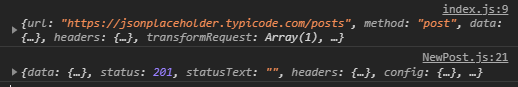
We can add header like that.

Interceptor acts as a middle man and all request and response will be pass this interceptor and move to the server or other component.





On both we can see that it executes from index.js:9 file. This shows interceptors.



And when sending a post request, it comes to interceptor and then moves to the NewPost component.

axios.interceptors.request.use(

  (request) => {

    console.log(request);

    // edit request config

    return request;                 // returning back the response else will have error.

  },

(error) => {

    console.log(error);

    return Promise.reject(error);   // returning back the response else will have error.

  }

);

This error is related to sending the request, for eg: no internet connectivity, so if the request sending fails.

If we return nly we can use that in then block in axios.get() etc.

Interceptor.request() – commonly we use, when add some common header like authorization

axios.interceptors.response.use(

  (response) => {

    console.log(response);

    // edit request config

    return response; // returning back the response else will have error.

  },

  (error) => {

    console.log(error);

    return Promise.reject(error); // returning back the response else will have error.

  }

);

This for the response, the response come to the interceptor then go to the component which needs it.



Response data or array(100) is coming from the server and passing to index.js initially to the interceptor, we can manipulate here and can send to the component.

**(xiii) Removing interceptors:**

You learned how to add an interceptor, getting rid of one is also easy. Simply store the reference of the interceptor in a variable and call eject  with that reference as an argument, to remove it (more info: <https://github.com/axios/axios#interceptors>):

1. var myInterceptor = axios.interceptors.request.use(function () {/\*...\*/});
2. axios.interceptors.request.eject(myInterceptor);

**(xiv) Setting a default global configuration for axios:**

Sometimes we don’t want to intercept the request but you want to setup soe global configuration.

axios.defaults.baseURL = "https://jsonplaceholder.typicode.com";

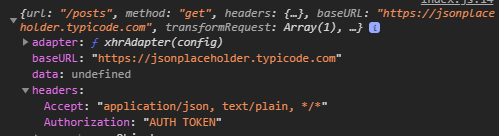
global configuration to set base url.

 axios.get("/posts")

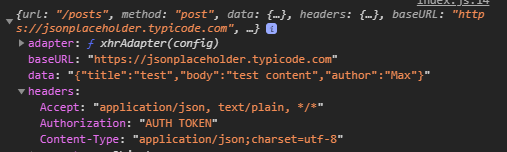
and this is enough only giving “/posts” the base url is joined with this and sent.

axios.defaults.headers.common["Authorization"] = "AUTH TOKEN";

we can set the common headers,



We can see the Authorization: “AUTH TOKEN” in request.



Post request sent with authorization and and content-type.

**(xv) Crating and using axios instances:**

Axios.create() can creates an instance of the axios.

Thorugh this instance, we can use axios globally and also where we need it.

Global:

axios.defaults.baseURL = "https://jsonplaceholder.typicode.com";

axios.defaults.headers.common["Authorization"] = "AUTH TOKEN";

axios.defaults.headers.post['Content-Type'] = 'application/json';

own axios for other users:

import axios from "axios";

const instance = axios.create({

  baseURL: "https://jsonplaceholder.typicode.com",

});

instance.defaults.headers.common["Authorization"] = "AUTH TOKEN FROM INSTANCE";

// instance.interceptors.request...

export default instance;

Blog.js

import axios from '../../axios';

here in blog, we use custom axios and other than blog component, we use global axios.

**(xvi) Useful resources & links:**

* Axios Docs: <https://github.com/axios/axios>

**SECTION 10: BURGER BUILDER PROJECT: ACCESSING A SERVER:**

**(i) Module intro:**

Creating a backend using firebase to store and get data from server db.

We should use realtime database and not firestore.

**(ii) Creating the axios instance:**

import axios from 'axios';

const instance = axios.create({

    baseURL: 'https://react-my-burger-a0df9-default-rtdb.firebaseio.com/'

});

export default instance;

**(iii) Sending a POST request:**

Firebase to function correctly we need to give .json at end.

purchaseContinueHandler = () => {

    // alert("You Continue!");

    // baseURL +

    const order = {

      ingredients: this.state.ingredients,

      price: this.state.totalPrice,

      customer: {

        name: "Sanjay",

        address: {

          no: "103",

          street: "4th street, krs nagar",

          pincode: "632007",

          state: "TN",

          country: "India",

        },

        email: "ksanjay00197@gmail.com",

        phoneNo: "8428066713",

      },

      deliveryMethod: "fastest",

    };

    axios

      .post("/orders.json", order)

      .then((response) => console.log(response))

      .catch((error) => console.log(error));

  };

import axios from 'axios';

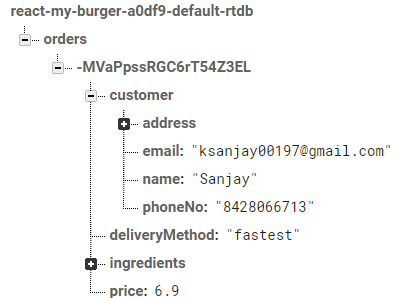
const instance = axios.create({

    baseURL: 'https://react-my-burger-a0df9-default-rtdb.firebaseio.com/'

});

export default instance;

below is data stored in firebase.



**(iv) Displaying a spinner while sending a request:**

**Spinner.js**

const spinner = () => <div className={classes.Loader}>Loading...</div>;

let orderSummary = (

      <OrderSummary

        ingredients={this.state.ingredients}

        purchaseCancelled={this.purchaseCancelHandler}

        purcharseContinued={this.purchaseContinueHandler}

        price={this.state.totalPrice}

      />

    );

    if (this.state.loading) {

      orderSummary = <Spinner />;

    }

purchaseContinueHandler = () => {

    this.setState({ loading: true });

    // alert("You Continue!");

    // baseURL +

    const order = {

      ingredients: this.state.ingredients,

      price: this.state.totalPrice.toFixed(2),

      customer: {

        name: "Sanjay",

        address: {

          no: "103",

          street: "4th street, krs nagar",

          pincode: "632007",

          state: "TN",

          country: "India",

        },

        email: "ksanjay00197@gmail.com",

        phoneNo: "8428066713",

      },

      deliveryMethod: "fastest",

    };

    axios

      .post("/orders.json", order)

      .then((response) => {

        this.setState({ loading: false, purchasing: false });

      })

      .catch((error) => {

        this.setState({ loading: false, purchasing: false });

      });

  };

**(v) Handling errors:**

const withErrorHandler = (WrappedComponent, axios) => {

  return class extends Component {

    state = {

      error: null,

    };

    componentDidMount() {

      axios.interceptors.request.use((req) => {

        this.setState({ error: null });

        return req;

      });

      axios.interceptors.response.use(res => res, (error) => {

        this.setState({ error: error });

      });

    }

    errorConfirmedHandler = () => {

      this.setState({ error: null });

    };

    render() {

      return (

        <Aux>

          <Modal show={this.state.error} modalClosed={this.errorConfirmedHandler}>

            {this.state.error ? this.state.error.message : null}

          </Modal>

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

        </Aux>

      );

    }

  };

};

Acts as higher order component.

**BurgerBuilder.js**

export default withErrorHandler(BurgerBuilder, axios);

this wraps the burger builder totally and any errors this will take care.

**(vi) Retrieving data from the backend:**

componentDidMount() {

    axios

      .get("/ingredients.json")

      .then((response) => {

        this.setState({ ingredients: response.data });

      })

      .catch((error) => this.setState({ error: true }));

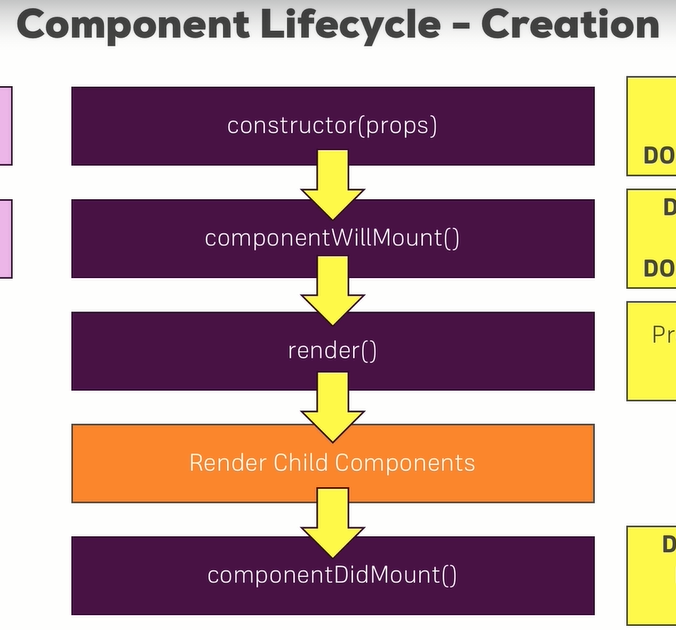
  }

This will get the data from ingredints with the baseURL and the state is set with the data. This is shown in build controls and also in state.

**What happens if we do not give .json?**

Wantedly made an error in url missing .json and error modal should be popped with backdrop. But it doesn’t even though we cover the whole burger builder with withErrorHandler().

Because withErrorHandler() we access the errors in componentDidMount(). In lifecycle,



componentDidMOunt() will call after rendering all child components, that why the error modal is not shown. Because the child components are render and we do set the error in componentDidMount and it finds no errors so nly.

So to show the modal with error, we have to use componentWillMount() to set the inteceptors to set the error state. But this lifecycle method is deprecated.

So another thing as instructor said we can use it in constructor(), but in constructor we can’t use setState, we can only initialize in constructor. So this doesn’t work.

And finally I used in render() which works fine. I should set axios before the child is rendered to show this error modal.

Solution: componentWillMount or render

**withErrorHandler.js**

componentWillMount() {

      axios.interceptors.request.use((req) => {

        this.setState({ error: null });

        return req;

      });

      axios.interceptors.response.use(

        (res) => res,

        (error) => {

          this.setState({ error: error });

        }

      );

    }

or

render() {

      axios.interceptors.request.use((req) => {

        this.setState({ error: null });

        return req;

      });

      axios.interceptors.response.use(

        (res) => res,

        (error) => {

          this.setState({ error: error });

        }

      ); // error is set before we render, then only modal class can show the error.

      return (

        <Aux>

          <Modal

            show={this.state.error}

            modalClosed={this.errorConfirmedHandler}

          >

            {this.state.error ? this.state.error.message : null}

          </Modal>

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

        </Aux>

      );

**(vii) Removing old interceptors:**

withErrorHandler is created to use for other components as a hoc. But if we give this to other component, the componentWillMount will call again and again. This will cause memory leaks. So to solve, we have to clear this axios.interceptors, because when we start using multi page routing, we will wrap withErrorHandler with other page components, that time the axios which is already stored will be called again and again and creates memory leak. So we can use componentWillUnmount() to remove the interceptors.

componentWillMount() {

      this.reqInterceptor = axios.interceptors.request.use((req) => {

        this.setState({ error: null });

        return req;

      });

      this.resInterceptor = axios.interceptors.response.use(

        (res) => res,

        (error) => {

          this.setState({ error: error });

        }

      );

    }

    componentWillUnmount() {

      console.log("Will Unmount", this.reqInterceptor, this.resInterceptor);

      axios.interceptors.request.eject(this.reqInterceptor);

      axios.interceptors.response.eject(this.resInterceptor);

    }

To check this we will remove the BurgerBuilder component from App.js after 5 secnds it mounted.

 state = {

    show: true,

  };

  componentDidMount() {

    setTimeout(() => {

      this.setState({ show: false });

    }, 5000);

  }

  render() {

    return (

      <div>

        <Layout>{this.state.show ? <BurgerBuilder /> : null}</Layout>

      </div>

    );

Here we set time for 5seconds and make the burger to become false. And it will not show. So this will trigger the componentWillUnmount() method.

And cleared state and componentDidMount() in App.js for normal code.

**SECTION 11: MULTI-PAGE-FEELING IN A SINGLE-PAGE-APP: ROUTING**

**1. Module Introduction:**

Routing is not build to the core, we gonna use other library. We gonna use major func routing.

**2. Routing and SPA’s:**

Routing is about being able to show different pages to the users.

Users page, account page, admin page.

How it can happen?

We use same single html, but we use javascript to render different pages for different paths.

So we don’t really have different files but simply we re-render parts of that single page or the entire single page depending on which path the user navigated in our application. This is what routing is about.

Parsing this path, so the path after our domain and showing appropriate jsx or component code in our app.

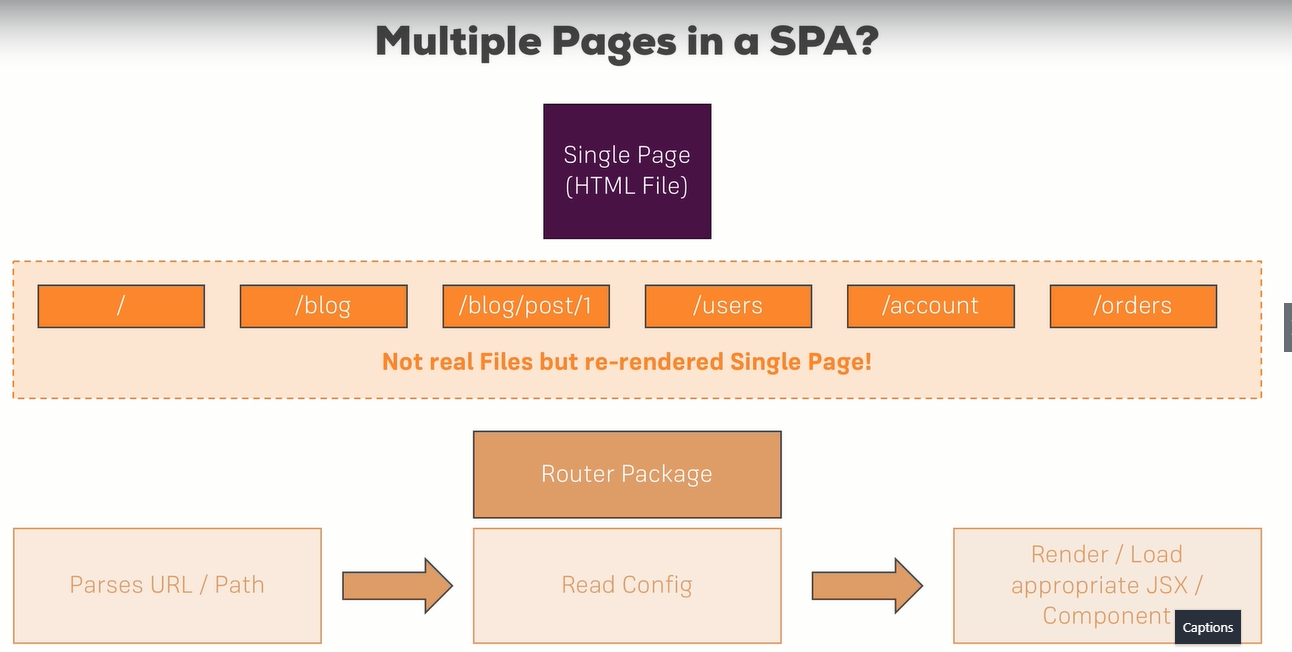
Two things in the routing lib:

a) it has to parse the URL path to understand where the user wanted to go to.

b) then developer has to configure different paths in our application which we support

and the router package can then read our configuration basically, so that it knows which paths are supported and what should happen when the user visits on of these paths.

c) then it render or load the jsx or component code depending on which path the user visited.



**(ii) Setting up links:**

<header>

          <nav>

            <ul>

              <li>

                <a href="/">Home</a>

              </li>

              <li>

                <a href="/new-post">New Post</a>

              </li>

            </ul>

          </nav>

        </header>

**(iii) Setting up the router package:**

**npm install –save react-router react-router-dom**

react-router – contains logic

react-router-dom – render it to screen

we have to enable the routing in index.js or app.js file to use it.

**App.js**

import { BrowserRouter } from "react-router-dom";

 <BrowserRouter>

        <div className="App">

          <Blog />

        </div>

      </BrowserRouter>

Now we can use routing anywhere inside this BrowserRouter component.

**(iv) react-router vs react-router-dom:**

We installed both react-router  and react-router-dom . **Technically, only react-router-dom  is required for web development**. It wraps react-router  and therefore uses it as a dependency.

We **don't need to install**react-router  on our own for it to work. You can omit this installation step, I left it in there for historic reasons and because I like to emphasize that the main package is named react-router. If you ever search for assistance, you probably want to search for "react router" - that's the name of the package.

**(v) Preparing the project for routing:**

Copy everything into newly created Posts container from Blog.js.

And make Blog.js to have only nav links and Posts component created.

class Blog extends Component {

  render() {

    console.log("[Blog.js] render");

    return (

      <div className="Blog">

        <header>

          <nav>

            <ul>

              <li>

                <a href="/">Home</a>

              </li>

              <li>

                <a href="/new-post">New Post</a>

              </li>

            </ul>

          </nav>

        </header>

        <Posts />

      </div>

**(vi) Setting up and rendering routes:**

<div className="Blog">

        <header>

          <nav>

            <ul>

              <li>

                <a href="/">Home</a>

              </li>

              <li>

                <a href="/new-post">New Post</a>

              </li>

            </ul>

          </nav>

        </header>

        <Route path="/" exact render={() => <Posts />} />

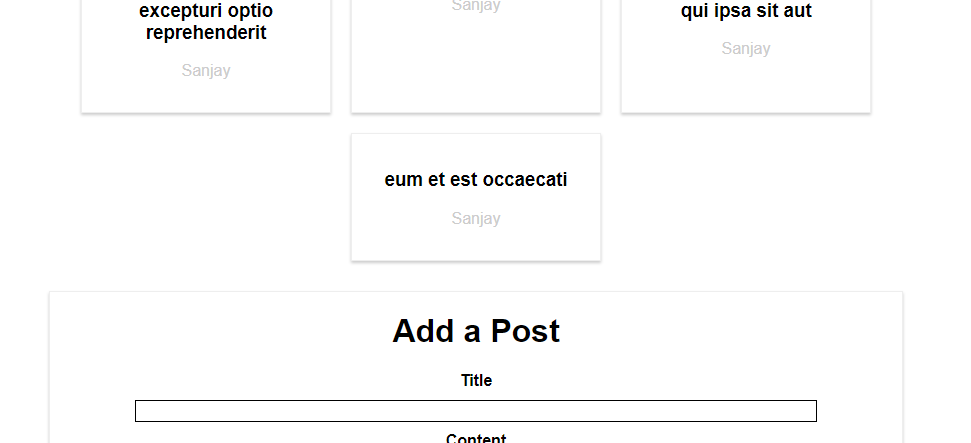
        <Route path="/new-post" exact render={() => <NewPost />} />

      </div>

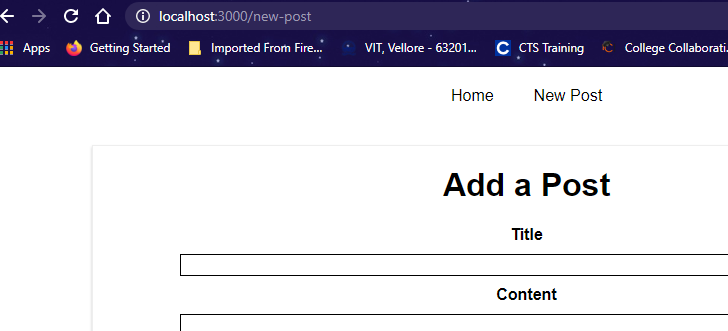
Here it makes to route that particular path to the particular Component.

And if we give exact then the path should be exactly the same to be rendered. Else if it is not the same path, then it wont view any component on that.

If we didn’t give exact, then it will render with the base path. For ex:



Here we can see both posts and new post in same in home and when routing to new-post,



We will again see this new post(add a post). Because in the code,

<Route path="/" exact render={() => <Posts />} />

<Route path="/" render={() => <NewPost />} />

Both have same path, so for that reason, in home path, the both component renders.

And in second image, in new posts we will see only the Newpost component, eventhough its path is not pointing to this, because when we click the other path, the router tries to find the path and its component, if it cant find the path, then there it will give this component, which has no exact attribute there.

If there is a exact keyword in this Route then it will render only to the exact path.

And when I tried to take exact from both things, then in home, we can see only what we expect, but when I moved to new-post, it renders both posts and new post, it happens because both route doesn’t has exact,

<Route path="/" render={() => <Posts />} />

        <Route path="/new-post" render={() => <NewPost />} />

And the second Route, the path is /new-post, for that it will first render the component for new-post, then it will render the / too, because both doesn’t have exact.

**Note:**

Giving **exact** is important.

**(vii) Rendering components for routes:**

 <Route path="/" exact component={Posts} />

 <Route path="/new-post" exact component={NewPost} />

Instead of render, we can also give component, and give their name to render it.

**(viii) Switching between pages:**

And when routing, we can see it loads a new page, and if this happens, the previous state will be lost. We don’t need to re-load the page, instead we only need to re-render the page to look like a new page.

In udemy, when we click pages, it loads the page but when we open a course and click other links in that, then we can find it doesn’t reload, here it just re-renders.

And the reload of page happens due to anchor tag present in the component.

 <a href="/">Home</a>

              </li>

              <li>

                <a href="/new-post">New Post</a>

**(ix) Using links to switch pages:**

So to overcome re-loading, we can use **Link** from react-router. With this the state is contained and not lost.

import { Route, Link } from "react-router-dom";

  <li>

             <Link to="/">Home</Link>

              </li>

              <li>

                <Link

                  to={{

                    pathname: "/new-post",

                    hash: "#submit",

                    search: "?quick-submit=true",

                  }}

                >

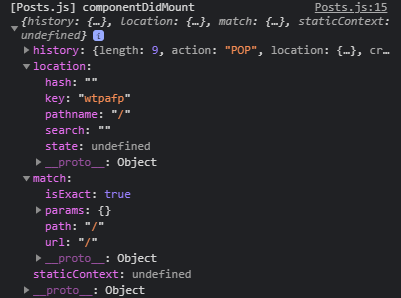
                  New Post

                </Link>

              </li>

Here we can also set, javascript object which has many attributes like, pathname, hash, search etc.

**(x) Using routing related props:**



we can print the props and can see what are all the props that are there in routing.

**(xi) The “withRouter” HOC & Route Props:**

Routing related props are not passed down to the component tree. When we print the props inside post component, we cannot see routing related props.

If we want to pass this routing related props to the child component, then we have two ways,

a) we can pass using spread operator as prop:

{...this.props} // all props

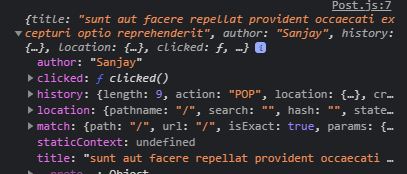
match={this.props.match} // particular prop

b) we can also a hoc on the post component - withRouter:

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

export default withRouter(post);

by doing like this, we can pass all the props here in the post component and we should use this in the post component itself to catch the routing related props.



Including the post props, we can also see the routing related props.

**(xii) Absolute vs Relative Paths:**

An absolute path is always appended to your domain.

example.com/paths, example.com/new-post…

if we give only, ‘/new-post’ it will always append to the root domain.

Relative path is like, takes the current path and append with those.

Example.com/posts (absolute path) -> example.com/posts/new-post (relative path)

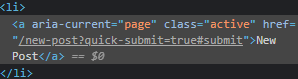
pathname: this.props.match.url + "/new-post"

**(xiii) Styling the active route:**

Instead of using Link, we can use NavLink to add styling to our active links.

import { Route, NavLink } from "react-router-dom";

<NavLink to="/">Home</NavLink>

 this aria-current and class is added, when we NavLink, so now we can style it.

After adding styling, we can that both links are active with the css and this happens because of the prefix in the “/” and “/new-post” in NavLink. And it is enough to add in first Home Navlink, because the New Post has no nested link to trigger active class.

So to solve the issue, we can add exact prop in NavLink.

We can also override the active class created using NavLink by a prop called **activeClassName.**

<NavLink to="/" activeClassName="my-active" exact>

<NavLink to="/" activeClassName="active" exact>

We can also set the styling in activeStyle prop.

<NavLink

                  to="/"

                  activeClassName="my-active"

                  activeStyle={{

                    color: "#fa923f",

                    textDecoration: "underline",

                  }}

                  exact

                >

                  Home

                </NavLink>

**(xiv) Passing route parameters:**

<Route path="/:id" exact component={FullPost} />   {/\* dynamic routing path \*/}

 <Link key={post.id} to={"/" + post.id}>

            <Post

              title={post.title}

              author={post.author}

              clicked={() => this.postSelectedHandler(post.id)}

            />

          </Link>

Making each post as a link and giving its route path. And when we click the Link, it will go to the path in Route and renders the FullPost component.

**(xv) Extracting route parameters:**

To extract the route parameter, we can see the props have a match prop, which has params props, and that params consists of all the parameters from the route path.

<Route path="/:id" exact component={FullPost} />

if (this.props.match.params.id) {

      if (

        !this.state.loadedPost ||

        (this.state.loadedPost && this.state.loadedPost.id !== this.props.id)

      ) {

        axios.get("/posts/" + this.props.match.params.id).then((response) => {

          // console.log(response);

          this.setState({ loadedPost: response.data });

        });

      }

This will now get the correct value from the backend and then shows to use. And we also changed the componentDidUpdate to Mount because before everything in one page, and full post is being updated, now its separate each time its mounted.

**(xvi) Passing query parameter & the fragment:**

You learned how to extract route parameters (=> :id  etc).

But how do you extract **search** (also referred to as "**query**") **parameters**(=> ?something=somevalue  at the end of the URL)? How do you extract the **fragment** (=> #something  at the end of the URL)?

**Query Params:**

You can pass them easily like this:

<Link to="/my-path?start=5">Go to Start</Link>

or

1. <Link
2. to={{
3. pathname: '/my-path',
4. search: '?start=5'
5. }}
6. >Go to Start</Link>

React router makes it easy to get access to the search string: props.location.search .

But that will only give you something like ?start=5

You probably want to get the key-value pair, without the ?  and the = . Here's a snippet which allows you to easily extract that information:

1. componentDidMount() {
2. const query = new URLSearchParams(this.props.location.search);
3. for (let param of query.entries()) {
4. console.log(param); // yields ['start', '5']
5. }
6. }

URLSearchParams  is a built-in object, shipping with vanilla JavaScript. It returns an object, which exposes the entries()  method. entries()  returns an Iterator - basically a construct which can be used in a for...of...  loop (as shown above).

When looping through query.entries() , you get **arrays** where the first element is the **key name** (e.g. start ) and the second element is the assigned **value** (e.g. 5 ).

#### ****Fragment:****

You can pass it easily like this:

<Link to="/my-path#start-position">Go to Start</Link>

or

1. <Link
2. to={{
3. pathname: '/my-path',
4. hash: 'start-position'
5. }}
6. >Go to Start</Link>

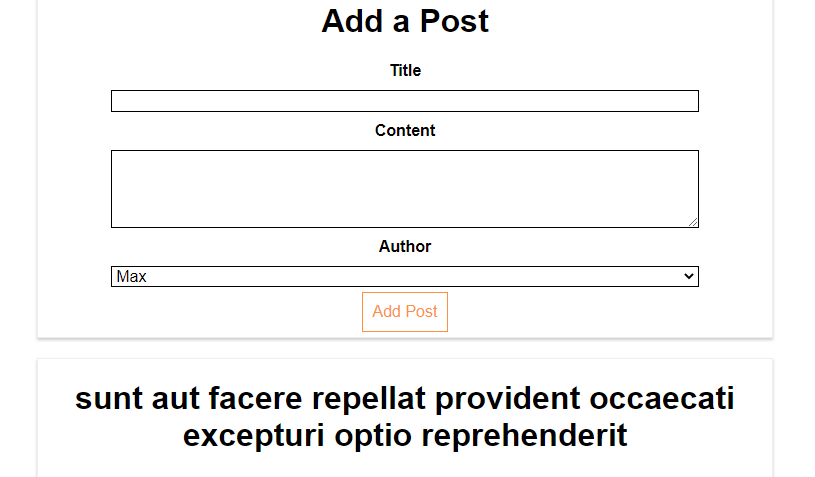
React router makes it easy to extract the fragment. You can simply access props.location.hash .

**(xvii) Using switch to load a single route:**

<Route path="/new-post" exact component={NewPost} />

<Route path="/:id" exact component={FullPost} />   {/\* dynamic routing path

This two has an issue like,



I can see both new-post and full-post in new-post route because of the route path what we define up. When I click the post, it should route to with the id, instead it takes new-post as id and renders below the new post.

**Solution 1:**

Blog.js

<Route path="/new-post" exact component={NewPost} />

<Route path="/posts/:id" exact component={FullPost} />

Posts.js

 <Link key={post.id} to={"/posts/" + post.id}>

            <Post

              title={post.title}

              author={post.author}

              clicked={() => this.postSelectedHandler(post.id)}

            />

          </Link>

Adding a separate route path which is viable.

**Solution 2:**

But sometimes we have to use only the given path like before, what should we do on that time,

We can use **Switch** from react-router-dom and wrap the other route and tells please route only one of the route that first one matches from given set of routes.

import { Route, NavLink, Switch } from "react-router-dom";

<Switch>

          <Route path="/" exact component={Posts} />

          <Route path="/new-post" exact component={NewPost} />

          <Route path="/:id" exact component={FullPost} />

        </Switch>

when using switch, order is very important.

<Switch>

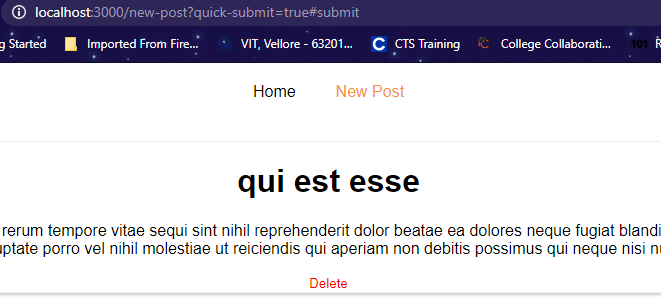
          <Route path="/" exact component={Posts} />

          <Route path="/:id" exact component={FullPost} />

          <Route path="/new-post" exact component={NewPost} />

          {/\* dynamic routing path \*/}

        </Switch>



Even in new-post route the full post component is rendered because of the order mismatch.

We can also mix and match the routes like, we can add route above the switch, inside switch and after the switch.

**(xviii) Navigating Programmatically:**

In history we have some methods like, push, pop, forward, backward we gonna make use of it and push the component to correct path in stack of path.

Forward is like, stack of path/page we can forward to that page that is top of stack and backward is like going back to the page.

// <Link key={post.id} to={"/" + post.id}>

          <Post

            key={post.id}

            title={post.title}

            author={post.author}

            clicked={() => this.postSelectedHandler(post.id)}

          />

          // </Link>

postSelectedHandler = (id) => {

    this.props.history.push({pathname: "/" + id});

    // this.props.history.push("/" + id);   // this will also work

  };

Instead of using Link, we can use like this and we can push the id to that path programmatically. And this renders as expected.

**(xix) Additional info regarding active links:**

When we click a post, home(changed to Posts) should be active because the post is belongs to that Posts but it isn’t. The reason is exact, since it treated as active if we exactly in that url. When we take exact from that, it acts as expected but the issue is, when we click new post, both posts and new post is active now, because exact is missing in posts.

Leaving it the same now, no issues.

**(xx) Understanding nested routes:**

We can use Route where ever we need in our project.

Posts.js

  <div>

        <section className="Posts">{posts}</section>

        <Route path="/:id" exact component={FullPost} />

      </div>

We took the Route from blog and put inside the post return which is a child component of Blog.

When we click the post, it should route to the route path which has its id.

<http://localhost:3000/1>

it moves here, but full post which is supposed to be visible here is not there.

We load the component Posts from Blog route, and this Posts has another route, its like route inside route.

<Switch>

          <Route path="/" exact component={Posts} />

          <Route path="/new-post" exact component={NewPost} />

        </Switch>

And it doesn’t show because in the blog route which acts as main “/” path has exact but there posts is not rendered.

Solution:

We can take the exact and change the order.

<Switch>

          <Route path="/new-post" component={NewPost} />

          <Route path="/posts" component={Posts} />

        </Switch>

<div>

        <section className="Posts">{posts}</section>

        <Route path="/posts/:id" component={FullPost} /> {/\* dynamic routing path \*/}

      </div>

Now its working but having issues,

i) we cant type posts like this everywhere, how to do dynamically

<Route path={this.props.match.url + "/:id"} component={FullPost} />

This is truly dynamically done.

**(xxi) Creating dynamic nested routes:**

ii) full post is not updating

reason is react-router behind the scenes, doesn’t replace the component all the time.

We have to implement componentDidUpdate to set the state. But when we set in componentDidUpdate, when the state changes it re-renders again and again then moves to infinite loop.

Solved by replacing **this.props.id** to **this.props.match.params.id**

But still there is infinite loop.

This happens because,

loadData() {

    if (this.props.match.params.id) {

      if (

        !this.state.loadedPost ||

        (this.state.loadedPost &&

          this.state.loadedPost.id !== this.props.match.params.id)

      ) {

        axios.get("/posts/" + this.props.match.params.id).then((response) => {

          // console.log(response);

          this.setState({ loadedPost: response.data });

        });

      }

    }

  }

The above line which is highlighted has one string and other one as number, so we check deep equality here and doesn’t pass and hence though the infinite loop exists.

To resolve this, we can use just equality instead of deep equality.

this.state.loadedPost.id !== +this.props.match.params.id

this.state.loadedPost.id !== parseInt(this.props.match.params.id)

this two things can convert a string to int.

**(xxii) Redirecting requests:**

To re-direct, we can use two ways,

a)

<Route path="/new-post" component={NewPost} />

<Route path="/posts" component={Posts} />

<Route path="/" component={Posts} />

Having the re-direct path and putting the same component.

b) other way is to take a redirect component from react-router-dom:

if re-direct is used outside of the Switch, then from cant be specified and always be redirect to “/posts”.

<Route path="/new-post" component={NewPost} />

<Route path="/posts" component={Posts} />

<Redirect from="/" to="/posts" />

{/\* <Route path="/" component={Posts} /> \*/}

Now its pretty fine.

**(xxiii) Conditional Redirects:**

In NewPost component, when we click the button it should re-direct and when posting the request using axios it should re-direct.

If we place Redirect outside of switch statement, we cant use from and should use only to.

<div className="NewPost">

        <Redirect to="/posts" />

        <h1>Add a Post</h1>

        <label>Title</label>

And if we give like this in jsx, then when clicking the New Post link, this will redirect it to the “/posts” because that is the first line in the new post, it will not render others instead it directly re-directs it to the posts component.

 let redirect = null;

    if (this.state.submitted) {

      redirect = <Redirect to="/posts" />;

    }

When submitted the data, after that redirect it to the posts component.

**(xxiv) Using the history prop to redirect (replace):**

If submitted the post successfully, we just want to switch the page, we can use the history prop on every component which is inside the BrowserRouter wrapped component.

axios.post("/posts", post).then((response) => {

      console.log(response);

      this.props.history.push('/posts');

      // this.setState({ submitted: true });

    });

We can use the push to redirect instead of Redirect component.

And push uses stack, and if we press the back button we go to the new post again.

While re-direct replaces the current page, we can go back but we cant go back to the last page.

We can also do the same thing with replace.

axios.post("/posts", post).then((response) => {

      console.log(response);

      this.props.history.replace("/posts");

      // this.props.history.push('/posts');

      // this.setState({ submitted: true });

    });

But now when we click the back button, it will not go back, because it replaces the stack of pages. It acts same like the redirect component.

So Redirect component and replace is nearly same.

**(xxv) Working with Guards:**

Navigation guards…

A guard is used when don’t know whether the user is authenticated or not. And there are some parts or routes in our app, we only want to allow the user to visit, if he is authenticate it.

 {this.state.auth ? (

            <Route path="/new-post" component={NewPost} />

          ) : null}

This will check the user is authenticated, if yes, then re-direct it to NewPost else null.

componentDidMount() {

    // if unauth => this.props.history.replace('/posts');

    console.log("NewPost", this.props);

  }

Or else, in the component(NewPost), which should be authenticated, we can also replace the current page to the other page, in componentDidMount.

**(xxvi) Handling the 404 Case:**

One way is to re-direct it to other path.

This should always come last.

<Switch>

          {this.state.auth ? (

            <Route path="/new-post" component={NewPost} />

          ) : null}

          <Route path="/posts" component={Posts} />

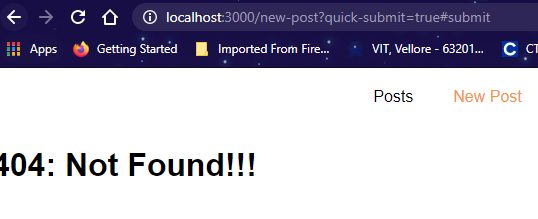
          <Route render={() => <h1>404: Not Found!!!</h1>} />

          {/\* <Redirect from="/" to="/posts" /> \*/}

          {/\* <Route path="/" component={Posts} /> \*/}

        </Switch>

2nd way to show a error component/page.



This wont work with re-direct. They both don’t work with together, because

{/\* <Redirect from="/" to="/posts" /> \*/}

This from=”/” acts as prefix and catches all the routes, so no matter even the order changes.

**(xxvii) Loading Routes Lazily:**

In network we can see, bundle.js kind of stuffs. In react when load the page, it loads everything and this bundle.js, which is compressed/optimized to use.

If it is a big project, with lots of features, even it has some pages, the user wont view at all.

Only load when it is needed i.e., lazy loading.

In main bundle, these component code will not be there only when clicking, we will dynamically lode this content from webpack.

Webpack will create a main bundle and dynamic chunk bundle for dynamic loading.

this is the chunk, when we click newpost.

import React, { Component } from "react";

const asyncComponent = (importComponent) => {

    return class extends Component {

        state = {

            component = null

        }

        componentDidMount() {

            importComponent().then(cmp => {

                this.setState({component: cmp.default})

            });

        }

        render() {

            const C = this.state.component;

            return C ? <C {...this.props} /> : null;

        }

    }

}

export default asyncComponent;

this is the async component created to be used and it is a hoc.

import asyncComponent from "../../hoc/asyncComponent";

// import NewPost from "./NewPost/NewPost";

const AsyncNewPost = asyncComponent(() => {

  return import("./NewPost/NewPost");

  // dynamic import syntax -> import(), this will imprt the component, only this anonymkus func executed

});

Deleting the import of NewPost and making it as asyncComponent.

And dynamically importing the NewPost component with their location. And this will loaded as chunk.js later.

<Switch>

          {this.state.auth ? (

            <Route path="/new-post" component={AsyncNewPost} />

          ) : null}

          <Route path="/posts" component={Posts} />

          <Route render={() => <h1>404: Not Found!!!</h1>} />

**(xxviii) Lazy loading with React Suspense:**

This works only in react 16.6.

class App extends Component {

  render() {

    return (

      <BrowserRouter>

        <React.Fragment>

          <nav>

            <NavLink to="/user">User Page</NavLink> |&nbsp;

            <NavLink to="/posts">Posts Page</NavLink>

          </nav>

          <Route path="/" component={Welcome} exact />

          <Route path="/user" component={User} />

          <Route path="/posts" component={Posts} />

        </React.Fragment>

      </BrowserRouter>

    );

  }

}

Gonna use lazy loading for User & Posts component.

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

import { BrowserRouter, Route, NavLink } from "react-router-dom";

// import Posts from './containers/Posts';

import User from "./containers/User";

import Welcome from "./containers/Welcome";

const Posts = React.lazy(() => import("./containers/Posts"));

class App extends Component {

  render() {

    return (

      <BrowserRouter>

        <React.Fragment>

          <nav>

            <NavLink to="/user">User Page</NavLink> |&nbsp;

            <NavLink to="/posts">Posts Page</NavLink>

          </nav>

          <Route path="/" component={Welcome} exact />

          <Route path="/user" component={User} />

          <Route

            path="/posts"

            render={() => (

              <Suspense fallback={<div>Loading...</div>}>

                <Posts />

              </Suspense>

            )}

          />

        </React.Fragment>

      </BrowserRouter>

    );

  }

}

export default App;

fallback is provided, if Posts may take time to load or not showing, we can instead show this loading msg, even it can be a spinner.

It is not like we should only use inside Route, we can also use it in some other places.

<React.Fragment>

        <button onClick={this.modeHandler}>Toggle</button>

        {this.state.showPosts ? (

          <Suspense fallback={<div>Loading...</div>}>

            <Posts />

          </Suspense>

        ) : (

          <User />

        )}

      </React.Fragment>

This will work only for client side rendering and not server side rendering.

**(xxix) Routing & The Server (Deployment):**

There is a problem, we will load all the files to server and server only will route or process the request.

The request from the user goes to server, and in server there is nothing like posts/1. This will result in 404 error. So whatever matters always forward the request to the React app, i.e., index.html.

Always load the index.html because react app only know the routes.

If we are calling like, example.com/my-app, we need to tell react about this, that the router needs to be precise. We need to set a base path.

<BrowserRouter basename='/my-app'>

We should give this browser route, if my-app/posts comes, it will search the posts accordingly.

Example: localhost:8080/zingerchat/login

Example: localhost:8080/my-app/posts

**(xxx) Useful resources:**

* React Router Docs: <https://reacttraining.com/react-router/web/guides/philosophy>

**SECTION 12: ADDING ROUTING TO OUR BURGER PROJECT:**

**1. Building the checkout container:**

First create a checkout summary with preview of burger

class Checkout extends Component {

  state = {

    ingredients: {

      salad: 1,

      meat: 1,

      cheese: 1,

      bacon: 1,

    },

  };

  render() {

    return (

      <div>

        <CheckoutSummary ingredients={this.state.ingredients} />

      </div>

    );

  }

}

const checkoutSummary = (props) => {

  return (

    <div className={classes.CheckoutSummary}>

      <h1>We hope it tastes well!</h1>

      <div style={{ width: "100%", margin: "auto" }}>

        <Burger ingredients={props.ingredients} />

      </div>

      <Button btnType="Danger" clicked>

        CANCEL

      </Button>

      <Button btnType="Success" clicked>

        CONTINUE

      </Button>

    </div>

  );

};

**2. Setting up routing & routes:**

**App.js**

class App extends Component {

  render() {

    return (

      <div>

        <Layout>

          <Switch>

            <Route path="/checkout" component={Checkout} />

            <Route path="/" exact component={BurgerBuilder} />

          </Switch>

        </Layout>

      </div>

    );

  }

}

**Index.js**

const app = (

  <BrowserRouter>

    <App />

  </BrowserRouter>

);

ReactDOM.render(app, document.getElementById("root"));

**3. Navigating to the checkout page:**

We can see the history and other props of Route in burger builder, but this cant be seen in burger component, which is the child of burger builder.

To get those props in the child component, we should use a hoc component of react-router-dom to get those props, i.e., withRouter method.

export default withRouter(burger); // includes the router props

**4. Navigating back & to next page:**

checkoutCancelledHandler = () => {

    this.props.history.goBack();

  };

  checkoutContinuedHandler = () => {

    this.props.history.replace("/checkout/contact-data");

  };

<CheckoutSummary

          ingredients={this.state.ingredients}

          checkoutCancelled={this.checkoutCancelledHandler}

          checkoutContinued={this.checkoutContinuedHandler}

        />

const checkoutSummary = (props) => {

  return (

    <div className={classes.CheckoutSummary}>

      <h1>We hope it tastes well!</h1>

      <div style={{ width: "100%", margin: "auto" }}>

        <Burger ingredients={props.ingredients} />

      </div>

      <Button btnType="Danger" clicked={props.checkoutCancelled}>

        CANCEL

      </Button>

      <Button btnType="Success" clicked={props.checkoutContinued}>

        CONTINUE

      </Button>

    </div>

  );

};

**5. Passing ingredients via Query params:**

**BurgerBuilder.js**

purchaseContinueHandler = () => {

const queryParams = [];

    for (let i in this.state.ingredients) {

      queryParams.push(

        encodeURIComponent(i) +

          "=" +

          encodeURIComponent(this.state.ingredients[i])

      );

    }

    const queryString = queryParams.join("&");

    this.props.history.push({

      pathname: "/checkout",

      search: "?" + queryString,

    });

  };

Checkout.js

componentDidMount() {

    const query = new URLSearchParams(this.props.location.search);

    const ingredients = {};

    for (let param of query.entries()) {

      // ['salad', '1', 'bacon', '1']

      ingredients[param[0]] = +param[1];

    }

    this.setState({ ingredients: ingredients });

  }

**6. Navigating to the contact data component:**

checkoutContinuedHandler = () => {

    this.props.history.replace("/checkout/contact-data");

  };

 <div>

        <CheckoutSummary

          ingredients={this.state.ingredients}

          checkoutCancelled={this.checkoutCancelledHandler}

          checkoutContinued={this.checkoutContinuedHandler}

        />

        <Route

          path={this.props.match.path + "/contact-data"}

          component={ContactData}

        />

      </div>

**7. Order submission & passing data between page:**

**contactData.js**

render() {

    let form = (

      <form>

        <input

          type="text"

          className={classes.Input}

          name="name"

          placeholder="Your Name"

        />

        <input

          type="email"

          className={classes.Input}

          name="email"

          placeholder="Your Mail"

        />

        <input

          type="text"

          className={classes.Input}

          name="street"

          placeholder="Street"

        />

        <input

          type="text"

          className={classes.Input}

          name="postal"

          placeholder="Postal Code"

        />

        <Button btnType="Success" clicked={this.orderHandler}>

          ORDER

        </Button>

      </form>

    );

    if (this.state.loading) {

      form = <Spinner />;

    }

    return (

      <div className={classes.ContactData}>

        <h4>Enter Your Contact Data</h4>

        {form}

      </div>

    );

Passing props from Checkout.js to ContactData.js

<Route

          path={this.props.match.path + "/contact-data"}

          render={(props) => (

            <ContactData

              ingredients={this.state.ingredients}

              price={this.state.totalPrice}

              {...props}

            />

          )}

        />

**8.Adding an orders page:**

**Order.js**

const order = (props) => (

  <div className={classes.Order}>

    <p>Ingredients: Salad (1)</p>

    <p>

      Price: <strong>USD 5.45</strong>

    </p>

  </div>

);

**Orders.js**

class Orders extends Component {

  render() {

    return (

      <div>

        <Order />

        <Order />

      </div>

    );

  }

}

**App.js**

Adding the route

 <Switch>

            <Route path="/checkout" component={Checkout} />

            <Route path="/orders" component={Orders} />

            <Route path="/" exact component={BurgerBuilder} />

          </Switch>

**9. Implementing navigation links:**

No need to check active or not, because navlink will automatically does this.

And we will take the active from NavigationItems to check whether active is working as NavLink is added.

Its adding, when we inspect we can see, but css style is not applied. Because css modules will change the class name in to unique ones. So active class will not be there.

To overcome this issue, we can use activeClassName of NavLink to use the style which we defined.

Now its working, but the issue is, both burger builder and orders are active as the style sets for those two links.

So to solve this, we can use exact attribute, to make each link as exactly single and so the styles are applied only to the exact links, its prefix is not taken only specific url will be taken.

**NavigationItems.js**

<ul className={classes.NavigationItems}>

    <NavigationItem link="/" exact>Burger Builder</NavigationItem>{" "}

    {/\* for boolean, we can give only active \*/}

    <NavigationItem link="/orders">Orders</NavigationItem>

  </ul>

**NavigationItem.js**

 <li className={classes.NavigationItem}>

    <NavLink

      to={props.link}

      exact={props.exact}

      activeClassName={classes.active}

    >

      {props.children}

    </NavLink>

  </li>

**10. Fetching Orders:**

class Orders extends Component {

  state = {

    orders: [],

    loading: true,

  };

  componentDidMount() {

    axios

      .get("/orders.json")

      .then((res) => {

        const fetchedOrders = [];

        for (let key in res.data) {

          fetchedOrders.push({

            ...res.data[key],

            id: key,

          });

        }

        this.setState({ loading: false, orders: fetchedOrders });

      })

      .catch((err) => {

        this.setState({ loading: false });

      });

  }

  render() {

    return (

      <div>

        <Order />

        <Order />

      </div>

    );

  }

}

export default withErrorHandler(Orders, axios);

**11. Outputting orders:**

To convert a string to number, we can use Number.parseFloat() or + sign before the string value.

 const ingredients = [];

  for (let ingredientName in props.ingredients) {

    ingredients.push({

      name: ingredientName,

      amount: props.ingredients[ingredientName],

    });

  }

  const ingredientOutput = ingredients.map((ig) => {

    return (

      <span

        style={{

          textTransform: "capitalize",

          display: "inline-block",

          margin: "0 8px",

          border: "1px solid #ccc",

          padding: "5px",

        }}

        key={ig.name}

      >

        {ig.name} ({ig.amount})

      </span>

    );

  });

  return (

    <div className={classes.Order}>

      <p>Ingredients: {ingredientOutput}</p>

      <p>

        Price: <strong>USD {Number.parseFloat(props.price).toFixed(2)}</strong>

      </p>

    </div>

  );

};

**SECTION 13: FORMS & FORM VALIDATON:**

**1. Module Introduction:**

We gonna see, forms and forms validation.

**2. Analyzing the app:**

Analyzing which data need to be updated/used.

**3. Creating a custom dynamic input component:**

const input = (props) => {

  let inputElement = null;

  switch (props.inputtype) {

    case "input":

      inputElement = <input className={classes.InputElement} {...props} />;

      break;

    case "textarea":

      inputElement = <textarea className={classes.InputElement} {...props} />;

      break;

    default:

      inputElement = <input className={classes.InputElement} {...props} />;

  }

  return (

    <div className={classes.Input}>

      <label className={classes.Label}>{props.label}</label>

      {inputElement}

    </div>

  );

};

ContactData.js

<form>

 <Input inputtype="input" type="text" name="name" placeholder="Your Name" />

 <Input inputtype="input" type="email" name="email" placeholder="Your Mail" />

 <Input inputtype="input" type="text" name="street" placeholder="Street" />

 <Input inputtype="input" type="text" name="postal" placeholder="Postal Code" />

 <Button btnType="Success" clicked={this.orderHandler}>ORDER</Button>

</form>

**4. Setting up a JS config for the FORM:**