

React

0. How to set up react development environment?

```
C:\Users\758370> node --version
```

v10.15.3

```
C:\Users\758370>npm --version
```

6.4.1

```
C:\Users\758370> d:
```

```
D:\> mkdir workspace
```

```
D:\> cd workspace
```

Now you can install your first node package with npm on the command line. You will install it globally with a -g flag. Because of installing it globally, you will always have access to it on the command line. Later on, when you install a node package for your project without the -g flag, you will only have access to the node package (node module) in your project. On the command line (in Visual Studio Code), type the following command to install create-react-app. Alternatively use node.js command prompt.

This package allows you to bootstrap React applications with zero-configuration.

```
D:\workspace> npm install -g create-react-app
```

```
C:\Users\758370\AppData\Roaming\npm\create-react-app ->
```

```
C:\Users\758370\AppData\Roaming\npm\node_modules\create-react-app\index.js
```

```
+ create-react-app@2.1.8
```

```
added 63 packages from 20 contributors in 27.783s
```

```
D:\workspace> create-react-app --version
```

2.1.8

You can bootstrap your first React.js application on Windows. You can use create-react-app by passing the name of your application to it on the command line:

```
D:\workspace> create-react-app my-react-app
```

Creating a new React app in D:\workspace\my-react-app.

Installing packages. This might take a couple of minutes.

Installing react, react-dom, and react-scripts...

+ react-dom@16.8.4

+ react@16.8.4

+ react-scripts@2.1.8

added 1843 packages from 718 contributors and audited 36244 packages in 765.597s

found 63 low severity vulnerabilities

run `npm audit fix` to fix them, or `npm audit` for details

Success! Created my-react-app at D:\workspace\my-react-app

D:\workspace> cd my-react-app

D:\workspace\ my-react-app > npm start

Compiled successfully!

You can now view my-react-app in the browser.

Local: http://localhost:3000/

On Your Network: http://10.40.219.97:3000/

Note that the development build is not optimized.

To create a production build, use npm run build.

1. Getting Started

1.1. Introduction

Library for creating reactive and fast JavaScript driven web applications.

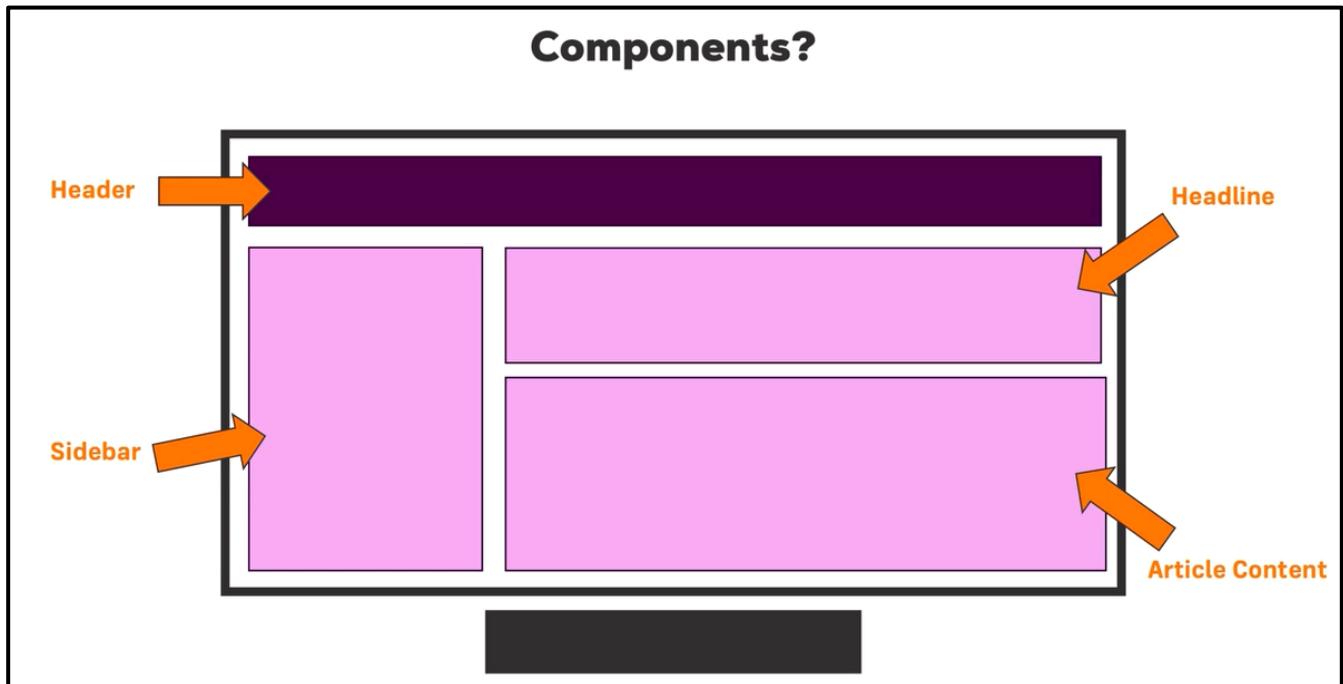
Users do not have to wait for page reload.

1.2. What is react?

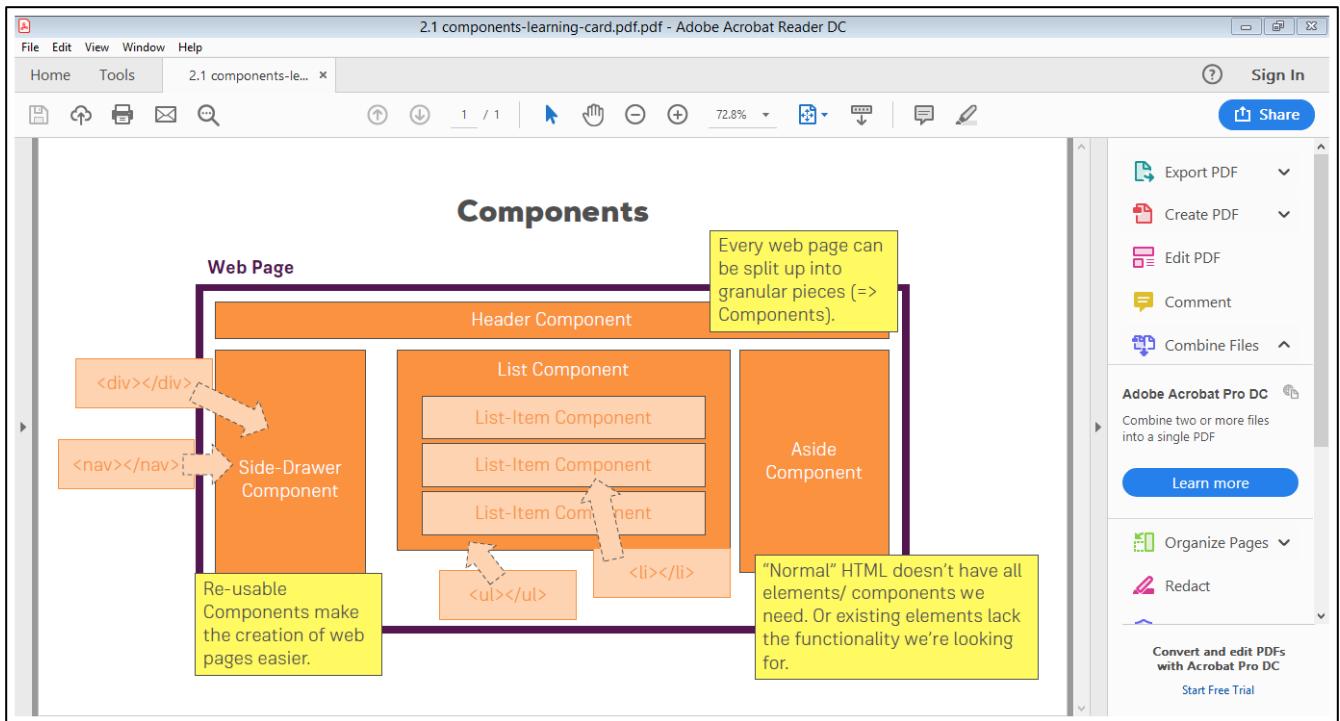
A JavaScript library for building user interfaces.

Components driven development.

React components are custom html elements.



1.2.1 components-learning-card



1.3. Real-World SPAs & React Web Apps

1.4. Adding the Right React Version to Codepen

1.5. Writing our First React Code

Using html and css (using codepen)

The screenshot shows a CodePen interface with two cards side-by-side.

Card 1 (Left):

- HTML:**

```
<h1>Max</h1>
<p>Your Age: 28</p>
</div>
```
- CSS:**

```
.person {
  display: inline-block;
  margin: 10px;
  border: 1px solid #eee;
  box-shadow: 0 2px 2px #ccc;
  width: 200px;
  padding: 20px;
}
```
- Output:**

Max

Your Age: 28

Card 2 (Right):

- HTML:**

```
<h1>Manu</h1>
<p>Your Age: 29</p>
</div>
```
- CSS:**

```
.person {
  display: inline-block;
  margin: 10px;
  border: 1px solid #eee;
  box-shadow: 0 2px 2px #ccc;
  width: 200px;
  padding: 20px;
}
```
- Output:**

Manu

Your Age: 29

Limitations:

- 1) We are using the same html code again and again. It will be even more problematic if we do something with these snippets in JavaScript.

Note: We can think of a person <div> as a component because the structure remains same, just the data inside varies. Here we can use react.

Add react library and JavaScript preprocessor (**Babel**) in Codepen JavaScript gear icon.

Pen Settings

HTML CSS **JavaScript** Behavior

JavaScript Preprocessor ?

Babel

Add External JavaScript ?

These scripts will run in this order and before the code in the JavaScript editor. You can also link to another Pen here, and it will run the JavaScript from it. Also try typing the name of any popular library.

≡ https://cdnjs.cloudflare.com/ajax/libs/react/15.6.1/react.min.js X

≡ https://cdnjs.cloudflare.com/ajax/libs/react/15.6.1/react-dom.min.js X

Quick-add: --- ▼ + add another resource

React is about the logics to create react components. **React DOM** is about rendering these components to the real DOM.

JavaScript preprocessor compiles the next gen JavaScript to the code that runs fine on the old browser (Where ES6 features are not supported). Write code with all the features we want but ship code with what works in the browser.

In its basic form, a react component is just a function.

Make sure that the component name starts with a capital letter.

HTML

```
<div id="p1"></div>
```

```
<div class="person">  
  <h1>Shyam</h1>  
  <p>Your age: 21</p>  
</div>
```

CSS

```
.person{  
  display: inline-block;  
  margin: 10px;  
  border: 1px solid #eee;  
  box-shadow: 0 6px 6px #ccc;  
  width: 200px;  
  padding: 20px  
}
```

JavaScript

```
// Person is a component and it has to return html which is to be rendered in DOM.  
  
function Person() {  
  
  // JSX syntax. HTML in JavaScript. It would not have worked had we not added Babel in our preprocessor. Babel  
  // allows us to use this special syntax (which looks like html but is not).  
  
  // JSX is just syntactical sugar, behind the scenes it gets compiled into normal JavaScript code.  
  
  return (  
    // react component that we imported is responsible for correctly parsing this JSX code.  
    // "class" is a keyword in JavaScript, so react turns this into className.  
  
    <div className="person">  
      <h1>Ram</h1>  
      <p>Your age: 20</p>  
    </div>  
  );  
  
  // To turn the above function into react component, we need to use react to render it to screen.  
  
  // render() allows us to render a JavaScript function as a component to the real DOM.
```

```
// The first argument is html element (what to render), 2nd argument takes where to render.
```

```
ReactDOM.render(<Person />, document.querySelector('#p1'));
```

The above code is not reusable though.

```
// Use {} to access dynamic content in react.
```

HTML

```
<div id="p1"></div>
```

```
<div id="p2"></div>
```

JavaScript

```
function Person(props) {
```

```
    return (
```

```
        <div class="person">
```

```
        // Use {} to access dynamic content in react
```

```
        <h1>{props.name}</h1>
```

```
        <p>Your age: {props.age} </p>
```

```
    </div>
```

```
);
```

```
}
```

```
// Here we can add attributes that will be bound to props in the function (Person) argument
```

```
ReactDOM.render (<Person name="Ram" age="20"/>,
```

```
    document.querySelector ('#p1'));
```

```
ReactDOM.render (<Person name="Shyam" age="21"/>,
```

```
    document.querySelector ('#p2'));
```

In the above code we wrote HTML (JSX) code once and reused it.

We are calling ReactDOM.render twice. Though there is nothing wrong with it, we can do it differently.

HTML

```
<div id="app"></div>
```

JavaScript

```
function Person(props){
```

```

return (
  <div class="person">
    <h1>{props.name}</h1>
    <p>Your age: {props.age}</p>
  </div>
);
}

// give any name

// This var will hold some JSX code

/* We have to wrap the content of this var in a root element (<div>). Bcz JSX has the requirement of only having one root element, so adjacent elements are not allowed */

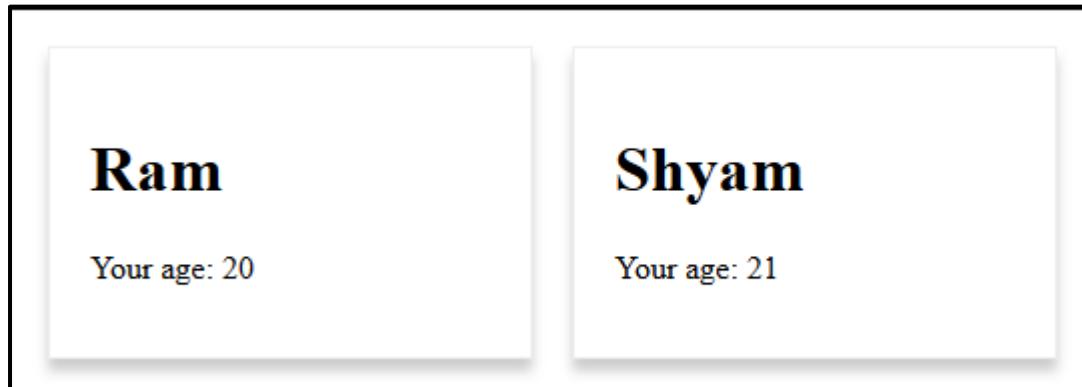
var app = (
  <div>
    <Person name="Ram" age="20" />
    <Person name="Shyam" age="21" />
  </div>
);

```

ReactDOM.render (app, document.querySelector ('#app'));

This is more popular way of creating react apps. With this approach you create single page applications.

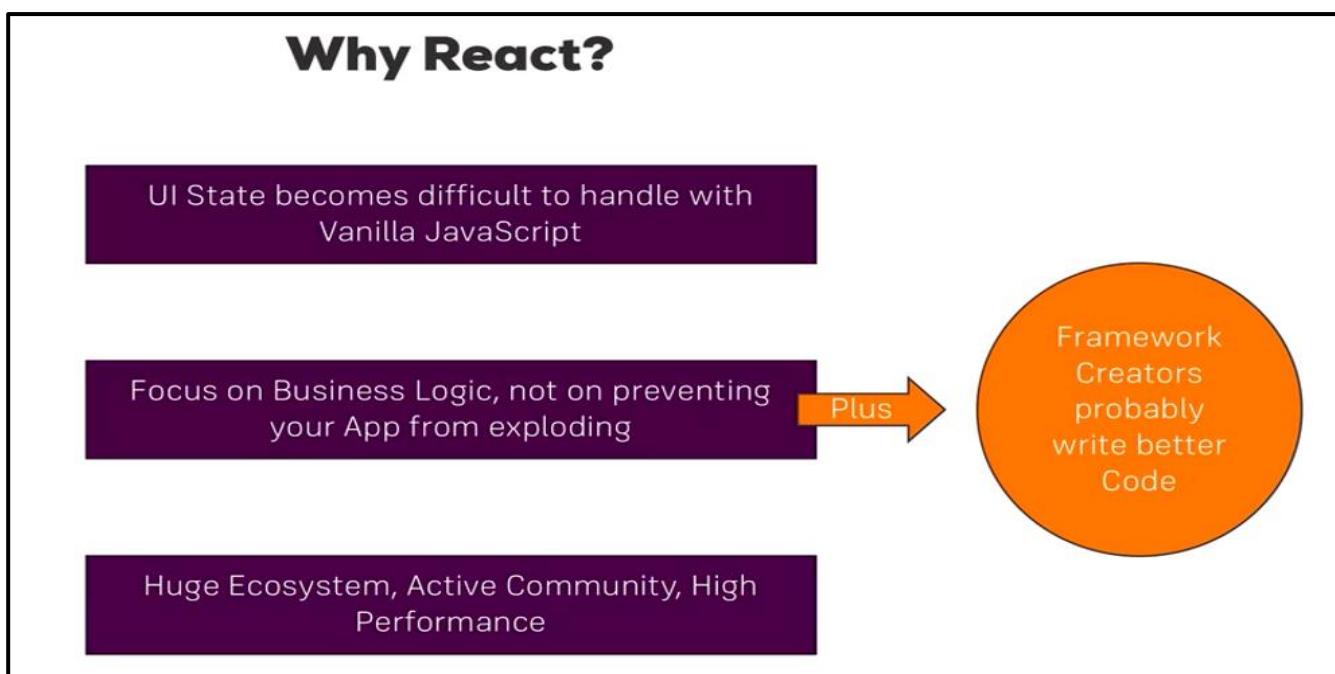
O/P:



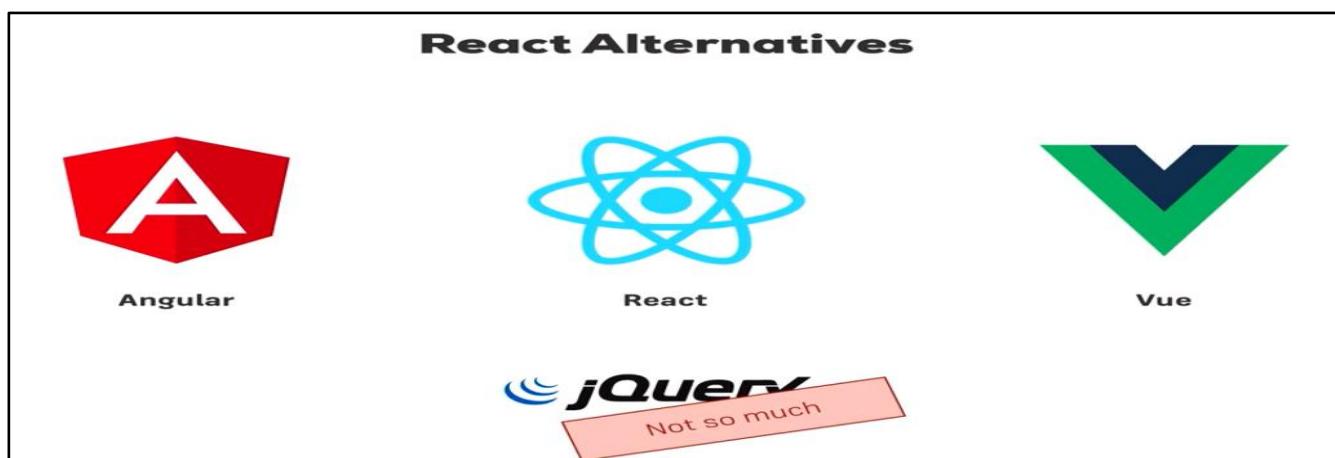
1.6. Why Should we Choose React

React helps us with the problems that we encounter with the normal JavaScript.

- 1) The UI state becomes difficult to manage. In bigger JavaScript applications, you have to manually target elements in your DOM and then if you change the structure of your code, chances are you need to change the way you targeted your element bcz you used `querySelector()`. Even if you use jQuery, traversing the DOM is easier, but it still is something that you have to keep in mind and if you got a complex web app where you dynamically add/remove elements, this can become cumbersome.
- 2) Allows us to focus on business logic.
- 3) Huge ecosystem



1.7. React Alternatives



1.8. Understanding Single Page Applications and Multi Page Applications

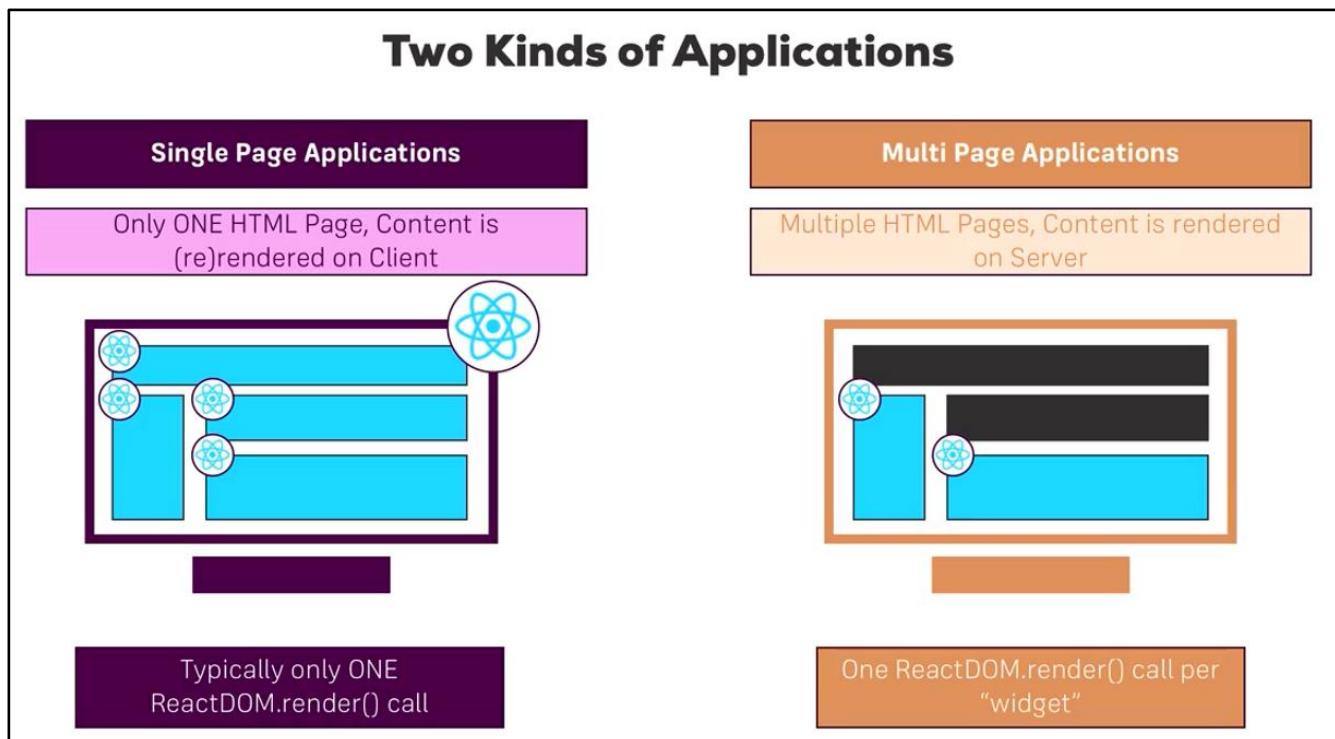
We can build single as well as multi-page applications using angular, react.

Single Page applications:

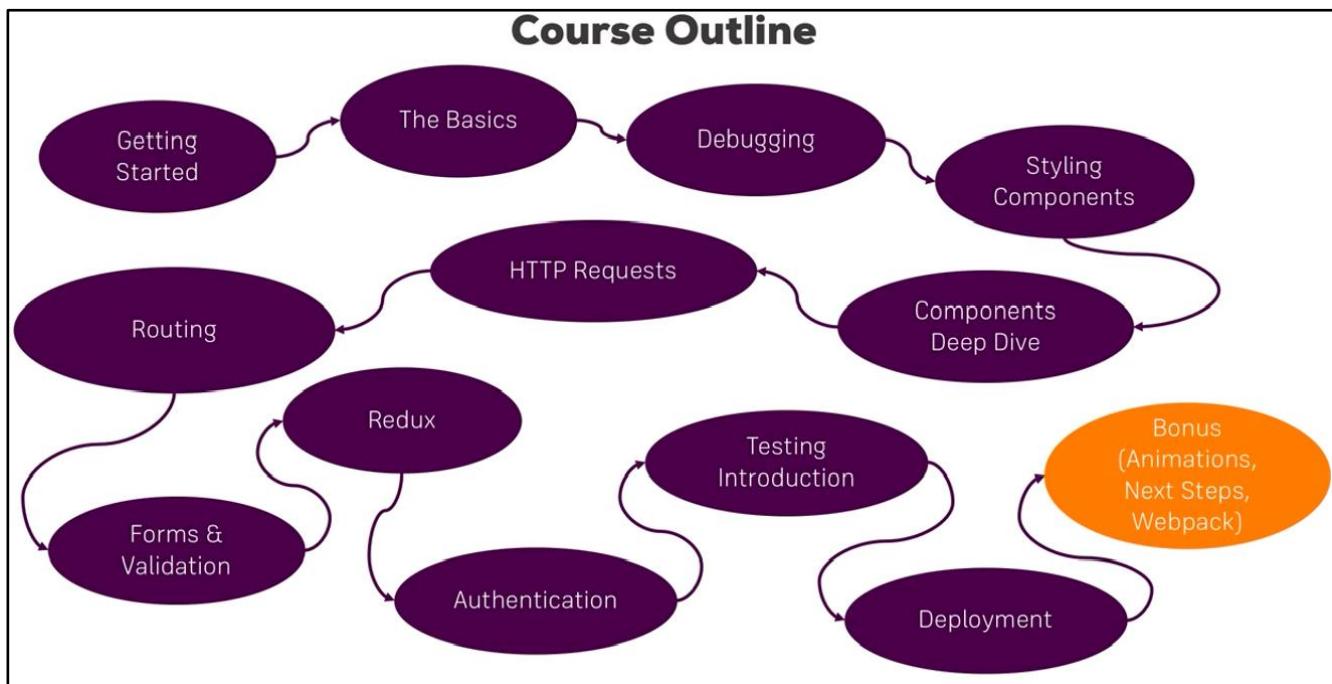
- We get back only one file by the server and we get back this file the first time user visits the page. Thereafter everything is managed with JavaScript with react. Entire page consists of components which are rendered and handled by JavaScript.
- Pages built up with components and every component is a react component and also the entire page is managed by a root react component and is under react's control.

Multi Page applications:

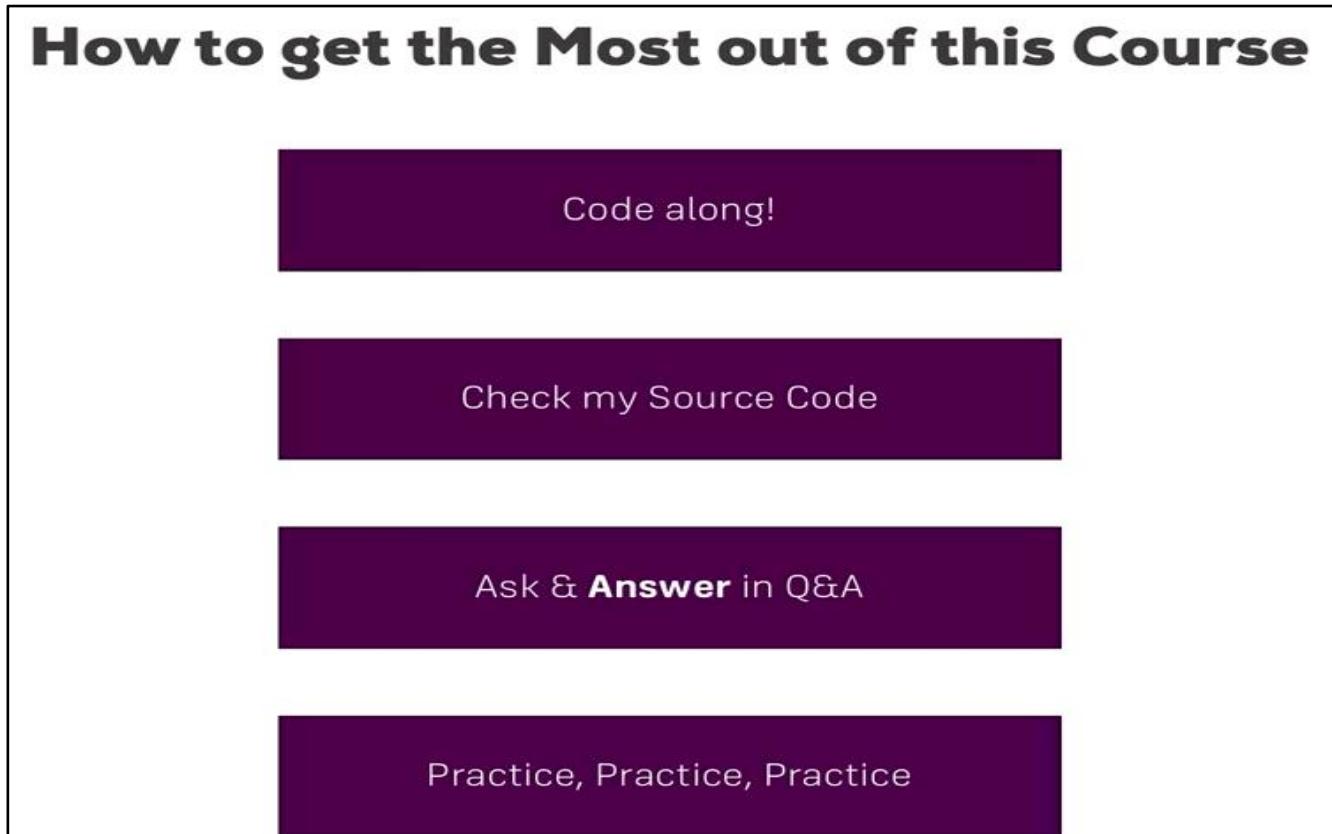
- We get back multiple HTML pages where each page has the content for the given route/url you visited. For example.com and example.com/users, we get back 2 different pages. In multipage applications we might also use React but only to create little widgets. So individually contained components we dump into that page but not the entire page is managed by react.
- Whereas in multipage, we can also theoretically split our app into multiple components but a lot of page is just going to be normal html, css code and some widgets (ex: an image gallery) we dump in that is managed by react. So the entire page is not under react control. The individual widgets do not know of each other's existence.



1.9. Course Outline



1.10. How to get the Most out of This Course



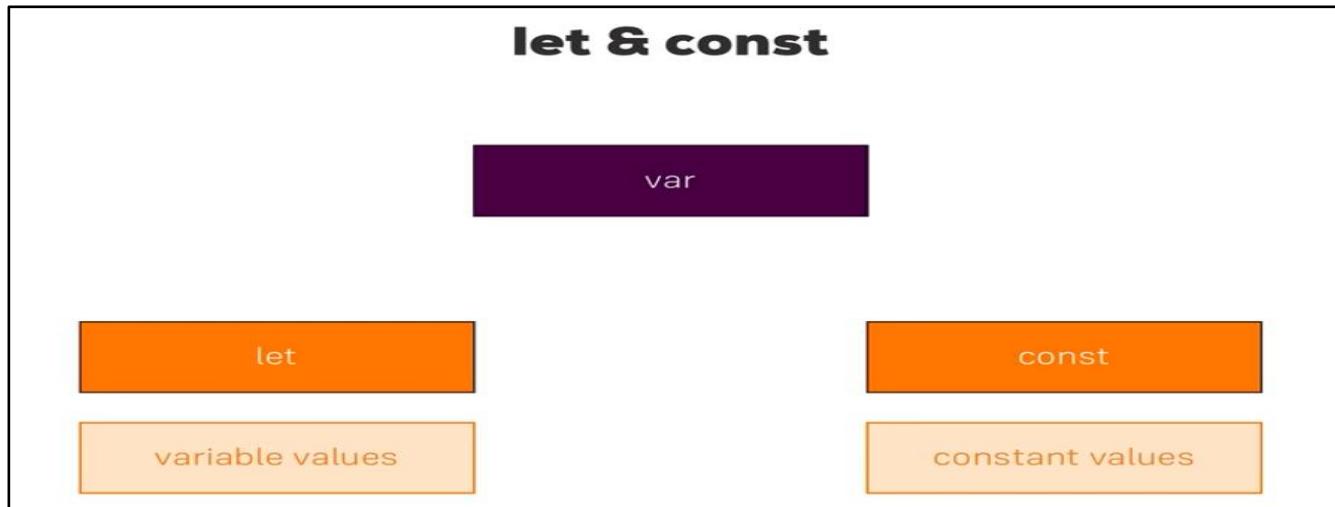
2. Refreshing Next Generation JavaScript (Optional)

2.1. Module Introduction

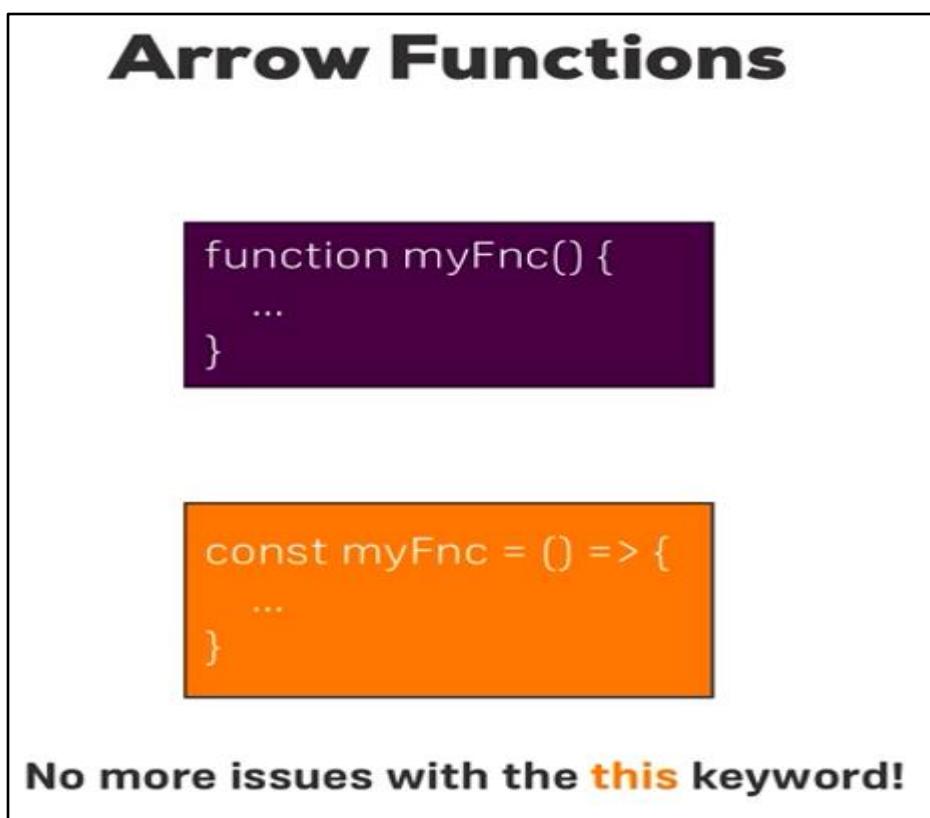
2.2. Understanding let and const

let → new var. Use let for the variable that is really a variable.

const → for constants



2.3. Arrow Functions



Note: In vanilla JavaScript ‘this’ keyword does not always refer to what you might have expected it to refer to. When you use ‘this’ in arrow function, it will always keep its context and will not surprisingly change at runtime.

Arrow Function → A different syntax for creating javascript functions.

```
function printMyName(name) {  
    console.log(name);  
}  
  
printMyName("Kunj");
```

// Arrow function with const

```
const printMyName1 = (name) => {  
    console.log(name);  
}  
  
printMyName1("Kunj Bihari");
```

// Arrow function with one argument. You can omit ()

```
const printMyName3 = name => {  
    console.log(name);  
}  
  
printMyName3("1 argument");
```

// Arrow function with zero argument. You have to put ()

```
const printMyName4 = () => {  
    console.log("Zero argument");  
}  
  
printMyName4();
```

```
// Arrow function with let  
  
let printMyName2 = (name) => {  
    console.log(name);  
}  
  
printMyName1("Kunj Bihari Singh");
```

/ if you have only one statement in the function body and that is return statement then you can omit {} and return keyword and make the function one liner.*

```
*/  
  
const multiply = (num) => {  
    return num*2;  
}  
  
console.log(multiply(2));
```

// The above function can be written as the below:

```
const multiply1 = num => num*2;  
  
console.log(multiply1(5));
```

O/P:

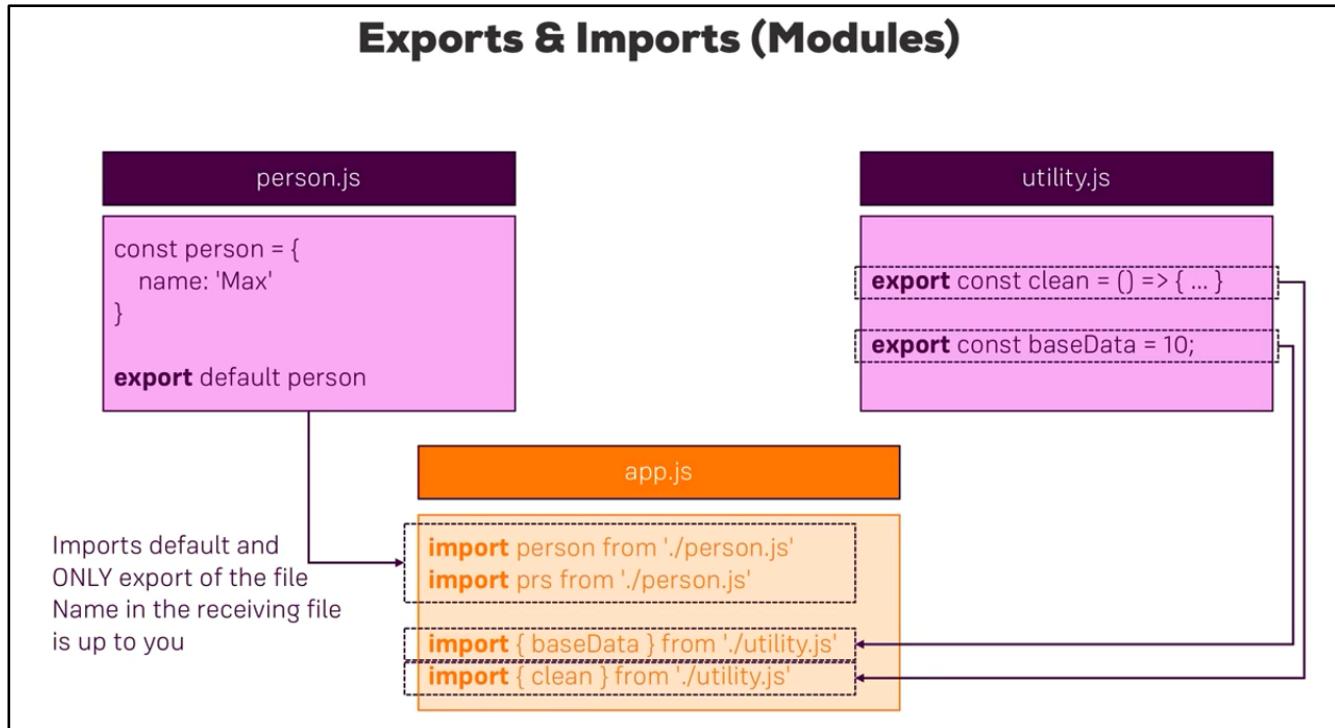
"Kunj"
"Kunj Bihari"

"1 argument"
"Zero argument"
"Kunj Bihari Singh"

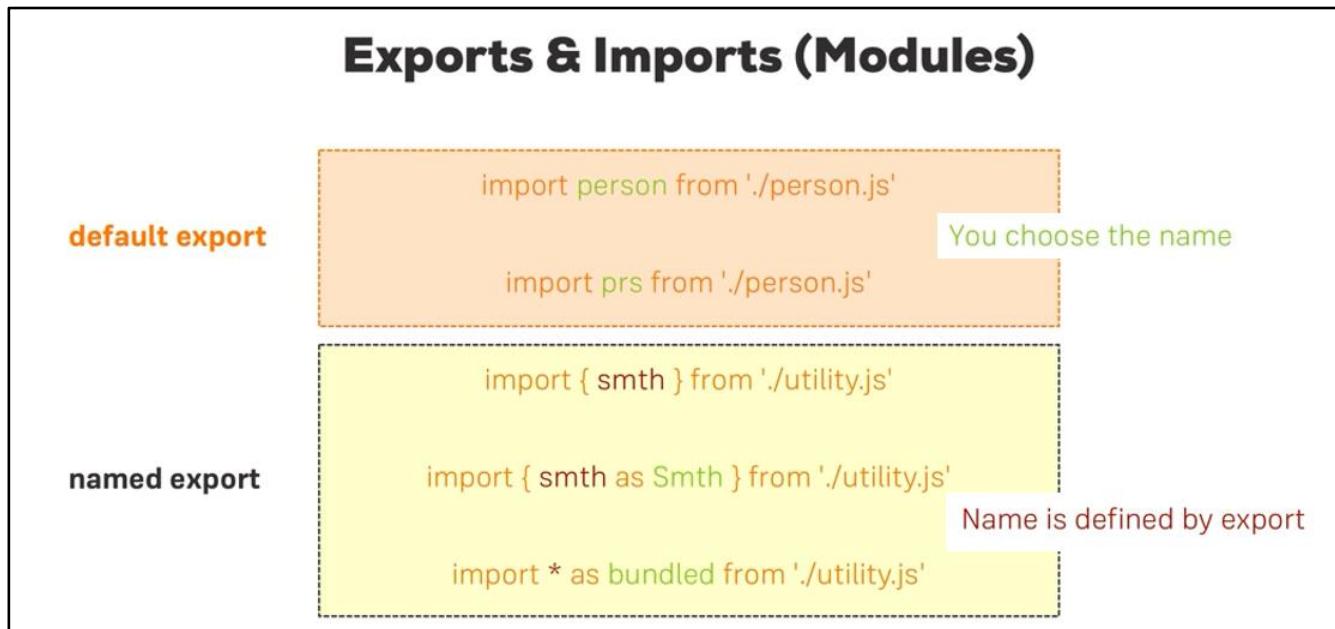
4
10

2.4. Exports and Imports

About writing modular code. Inside of a JavaScript file, we can import content from another file.



Default keyword marks the '**default export**' of a file. Whereas from utility.js the exports are called '**Named Exports**'.



2.5. Understanding Classes

Classes are blueprints for JavaScript objects.

Classes

Property class Person {
 name = 'Max'
 call = () => {...}
}

Method

Usage
(constructor functions anyone?)

```
const myPerson = new Person()  
myPerson.call()  
console.log(myPerson.name)
```

Inheritance
(prototypes anyone?)

```
class Person extends Master
```

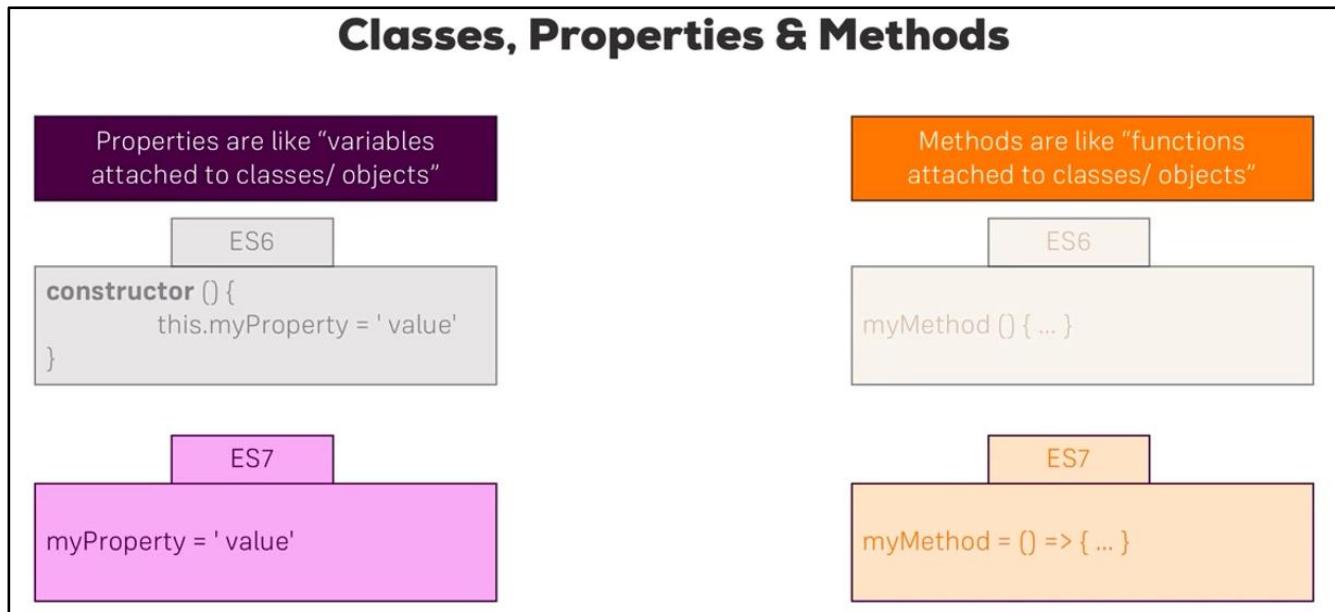
JavaScript ▾

```
class Human {  
  constructor() {  
    this.gender = 'male';  
  }  
  
  printGender() {  
    console.log(this.gender);  
  }  
}  
  
class Person extends Human {  
  constructor() {  
    super();  
    this.name = 'Max';  
    this.gender = 'female';  
  }  
  
  printMyName() {  
    console.log(this.name);  
  }  
}  
  
const person = new Person();  
person.printMyName();  
person.printGender();
```

Console
"Max"
"female"

2.6. Classes, Properties and Methods

Modern JavaScript way of writing properties and methods.



The screenshot shows a browser's developer tools console output. On the left, the code for `Human` and `Person` classes is displayed. On the right, the console output shows the execution of `person.printMyName()` and `person.printGender()`, with the results "Max" and "female" respectively.

```
ES6 / Babel ▾
class Human {
  gender = 'male';

  printGender = () => {
    console.log(this.gender);
  }
}

class Person extends Human {
  name = 'Max';
  gender = 'female';

  printMyName = () => {
    console.log(this.name);
  }
}

const person = new Person();
person.printMyName();
person.printGender();

Console
"Max"
"female"
```

2.7. The Spread & Rest Operator

Spread & Rest Operators

...

Spread

Used to split up array elements OR object properties

```
const newArray = [...oldArray, 1, 2]
const newObject = { ...oldObject, newProp: 5 }
```

Rest

Used to merge a list of function arguments into an array

```
function sortArgs(...args) {
  return args.sort()
}
```

Spread Operator:

ES6 / Babel ▾

```
const numbers = [1, 2, 3];
const newNumbers = [...numbers, 4];

console.log(newNumbers);
```

Console

```
[1, 2, 3, 4]
```

ES6 / Babel ▾

```
const person = {
  name: 'Max'
};

const newPerson = {
  ...person,
  age: 28
}

console.log(newPerson);
```

Console

```
[object Object] {
  age: 28,
  name: "Max"
}
```

Rest operator:

```
ES6 / Babel ▾  
const filter = (...args) => {  
  return args.filter(el => el === 1);  
}  
  
console.log(filter(1, 2, 3));
```

Console
[1]
➤

2.8. Destructuring

Difference between spread operator and destructuring:

Spread takes out all elements, properties and distributes them in a new array or object. Destructuring allows you to pull out single element/property and store them in a variable.

Destructuring

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

Array Destructuring

```
[a, b] = ['Hello', 'Max']  
console.log(a) // Hello  
console.log(b) // Max
```

Object Destructuring

```
{name} = {name: 'Max', age: 28}  
console.log(name) // Max  
console.log(age) // undefined
```

ES6 / Babel ▾

```
const numbers = [1, 2, 3];
[num1, , num3] = numbers;
console.log(num1, num3);
```

Console

1

3

2.9. Reference and Primitive Types Refresher

Primitive type:

// This is a primitive type

```
let number = 1;
```

// num2 copies the value from number

```
let num2 = number;
```

copies the value, not the reference

```
*/
```

```
number = 2;
```

```
console.log(num2); // o/p : 1
```

Note: Integer, Boolean, character are primitive types. Array and objects are reference types (here reference will be copied, not the value).

Reference type:

// person object is stored in the memory and 'person' is a reference to it

```
const person = {
```

```
  name: 'Kunj'
```

```
};
```

```
const secondPerson = person;
```

but it had not copied the 'person' value into 'secondPerson'.

The pointer 'person' will be copied to 'secondPerson'.

```
*/
```

```
person.name = "Ram";
```

'secondPerson' would have printed the value 'Kunj'.

*/

console.log(secondPerson);

O/P;

[object Object] {

 name: "Ram"

}

Q) How to actually copy the object value, not the reference?

Ans: Using spread operator

// person object is stored in the memory and 'person' is a reference to it

const person = {

 name: 'Kunj'

};

// Use of spread operator to copy object value

const secondPerson = {

 ...person

};

person.name = 'Ram';

console.log(secondPerson);

O/P:

[object Object] {

 name: "Kunj"

}

2.10. Refreshing Array Functions

const numbers = [1, 2, 3];

// double all the no's of numbers array and assign into new array

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

 return num * 2;

});

```
// Old array is unchanged  
console.log(numbers); // o/p: [1, 2, 3]  
console.log(doubleNumArray); // o/p: [2, 4, 6]
```

2.12. Next-Gen JavaScript – Summary

In this module, I provided a brief introduction into some core next-gen JavaScript features, of course focusing on the ones you'll see the most in this course. Here's a quick summary!

let & const

Read more about `let` : <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>

Read more about `const` : <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const>

`let` and `const` basically replace `var`. You use `let` instead of `var` and `const` instead of `var` if you plan on never re-assigning this "variable" (effectively turning it into a constant therefore).

ES6 Arrow Functions

Read more: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Arrow functions are a different way of creating functions in JavaScript. Besides a shorter syntax, they offer advantages when it comes to keeping the scope of the `this` keyword (see [here](#)).

Arrow function syntax may look strange but it's actually simple.

```
function callMe(name) {  
    console.log(name);  
}
```

which you could also write as:

```
const callMe = function(name) {  
    console.log(name);  
}
```

becomes:

```
const callMe = (name) => {
    console.log(name);
}
```

Important:

When having **no arguments**, you have to use empty parentheses in the function declaration:

```
const callMe = () => {
    console.log('Max!');
}
```

When having **exactly one argument**, you may omit the parentheses:

```
const callMe = name => {
    console.log(name);
}
```

When **just returning a value**, you can use the following shortcut:

```
const returnMe = name => name
```

That's equal to:

```
const returnMe = name => {
    return name;
}
```

Exports & Imports

In React projects (and actually in all modern JavaScript projects), you split your code across multiple JavaScript files - so-called modules. You do this, to keep each file/ module focused and manageable.

To still access functionality in another file, you need `export` (to make it available) and `import` (to get access) statements.

You got two different types of exports: **default** (unnamed) and **named** exports:

default => `export default ...;`

named => `export const someData = ...;`

You can import **default exports** like this:

```
import someNameOfYourChoice from './path/to/file.js';
```

Surprisingly, `someNameOfYourChoice` is totally up to you.

Named exports have to be imported by their name:

```
import { someData } from './path/to/file.js';
```

A file can only contain one default and an unlimited amount of named exports. You can also mix the one default with any amount of named exports in one and the same file.

When importing **named exports**, you can also import all named exports at once with the following syntax:

```
import * as upToYou from './path/to/file.js';
```

upToYou is - well - up to you and simply bundles all exported variables/functions in one JavaScript object. For example, if you `export const someData = ... (/path/to/file.js)` you can access it on `upToYou` like this: `upToYou.someData`.

Classes

Classes are a feature which basically replace constructor functions and prototypes. You can define blueprints for JavaScript objects with them.

Like this:

```
class Person {
  constructor () {
    this.name = 'Max';
  }
}

const person = new Person();
console.log(person.name); // 'Max'
```

In the above example, not only the class but also a property of that class (`=> name`) is defined. They syntax you see there, is the "old" syntax for defining properties. In modern JavaScript projects (as the one used in this course), you can use the following, more convenient way of defining class properties:

```
class Person {
  name = 'Max';
}

const person = new Person();
console.log(person.name); // prints 'Max'
```

You can also define methods. Either like this:

```
class Person {
  name = 'Max';
  printMyName () {
```

```

        console.log(this.name); // this is required to refer to the class!
    }
}

const person = new Person();
person.printMyName();

```

Or like this:

```

class Person {
    name = 'Max';
    printMyName = () => {
        console.log(this.name);
    }
}

const person = new Person();
person.printMyName();

```

The second approach has the same advantage as all arrow functions have: The `this` keyword doesn't change its reference.

You can also use **inheritance** when using classes:

```

class Human {
    species = 'human';
}

class Person extends Human {
    name = 'Max';
    printMyName = () => {
        console.log(this.name);
    }
}

const person = new Person();
person.printMyName();
console.log(person.species); // prints 'human'

```

Spread & Rest Operator

The spread and rest operators actually use the same syntax: ...

Yes, that is the operator - just three dots. Its usage determines whether you're using it as the spread or rest operator.

Using the Spread Operator:

The spread operator allows you to pull elements out of an array (=> split the array into a list of its elements) or pull the properties out of an object. Here are two examples:

```

const oldArray = [1, 2, 3];
const newArray = [...oldArray, 4, 5]; // This now is [1, 2, 3, 4, 5];

```

Here's the spread operator used on an object:

```
const oldObject = {
  name: 'Max'
};

const newObjet = {
  ...oldObject,
  age: 28
};
```

`newObject` would then be

```
{
  name: 'Max',
  age: 28
}
```

The spread operator is extremely useful for cloning arrays and objects. Since both are [reference types \(and not primitives\)](#), copying them safely (i.e. preventing future mutation of the copied original) can be tricky. With the spread operator you have an easy way of creating a (shallow!) clone of the object or array.

Destructuring

Destructuring allows you to easily access the values of arrays or objects and assign them to variables.

Here's an example for an array:

```
const array = [1, 2, 3];
const [a, b] = array;
console.log(a); // prints 1
console.log(b); // prints 2
console.log(array); // prints [1, 2, 3]
```

And here for an object:

```
const myObj = {
  name: 'Max',
  age: 28
}
const {name} = myObj;
console.log(name); // prints 'Max'
console.log(age); // prints undefined
console.log(myObj); // prints {name: 'Max', age: 28}
```

Destructuring is very useful when working with function arguments. Consider this example:

```
const printName = (personObj) => {
  console.log(personObj.name);
}
printName({name: 'Max', age: 28}); // prints 'Max'
```

Here, we only want to print the name in the function but we pass a complete person object to the function. Of course this is no issue but it forces us to call `personObj.name` inside of our function. We can condense this code with destructuring:

```
const printName = ({name}) => {
  console.log(name);
}
printName({name: 'Max', age: 28}); // prints 'Max'
```

We get the same result as above but we save some code. By destructuring, we simply pull out the `name` property and store it in a variable/ argument named `name` which we then can use in the function body.

JS Array Functions

Not really next-gen JavaScript, but also important: JavaScript array functions like `map()`, `filter()`, `reduce()` etc.

You'll see me use them quite a bit since a lot of React concepts rely on working with arrays (in immutable ways).

The following page gives a good overview over the various methods you can use on the array prototype - feel free to click through them and refresh your knowledge as required: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

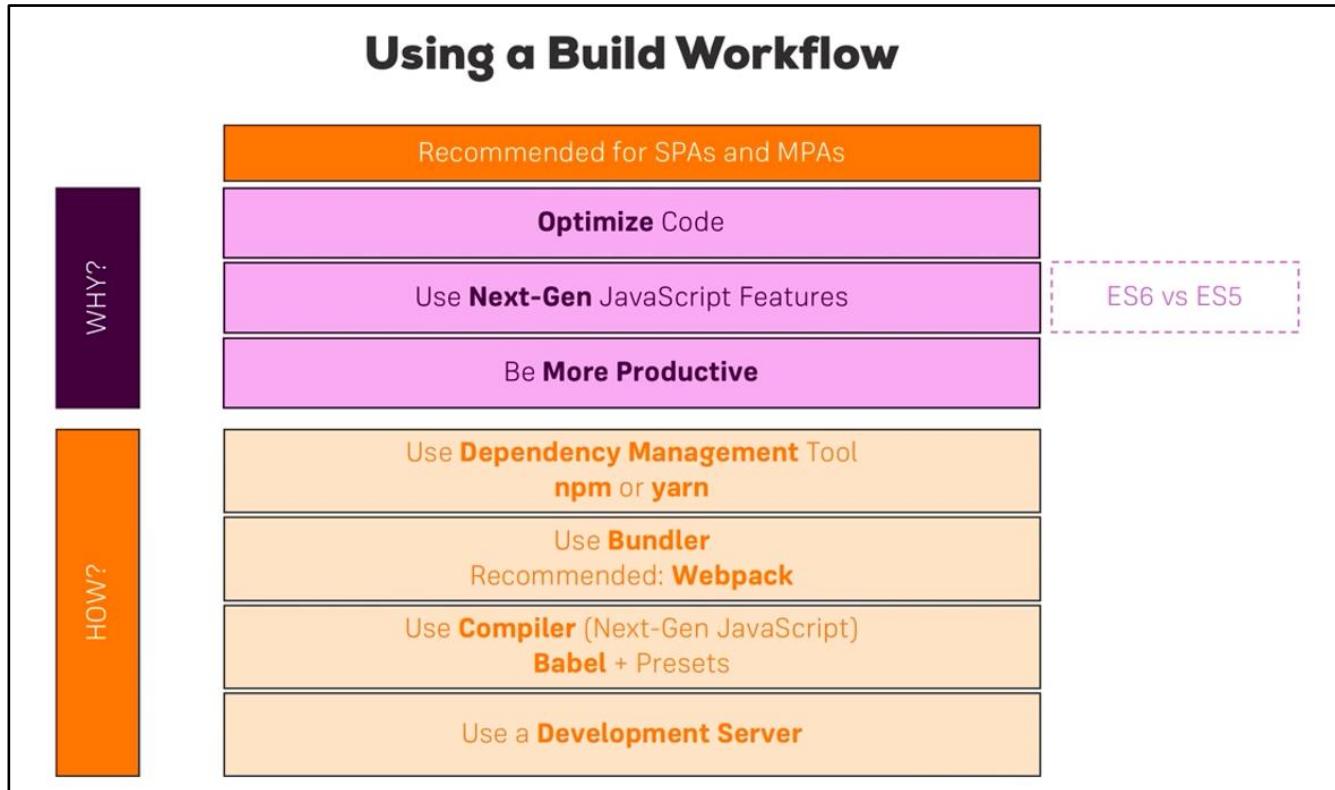
Particularly important in this course are:

- `map()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map
- `find()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find
- `findIndex()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex
- `filter()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter
- `reduce()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reduce?v=b
- `concat()` => [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array\(concat?v=b](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array(concat?v=b)
- `slice()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice
- `splice()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

3. Understanding the Base Features & Syntax

3.1. Module Introduction

3.2. The Build Workflow



3.3. Using create-react-app

3.4. Understanding the Folder Structure

a) Root folder

package-lock.json → locking in the dependency we are using

package.json → General dependencies are defined here

package.json

```
{  
  "name": "my-react-app",  
  "version": "0.1.0",  
  "private": true,  
  "dependencies": {  
    "radium": "^0.25.1",  
    "react": "^16.8.4",  
    "react-dom": "^16.8.4",  
    "react-scripts": "2.1.8"  
  }  
}
```

```

},
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
"eslintConfig": {
  "extends": "react-app"
},
"browserslist": [
  ">0.2%",
  "not dead",
  "not ie <= 11",
  "not op_mini all"
]
}

```

“scripts” → run these scripts with ***npm run <script name>***

npm run start = npm start

start: starts development server. Watches code. Compiles, optimizes code

build: Once you are ready to deploy your code, run ‘***npm run build***’ to optimize code even more and save in a folder

b) node_modules

Holds dependencies and sub-dependencies of our project

c) public

root folder which gets served by the web server at the end.

Index.html → Single html page for the project. We will never add anymore html file. If you are creating a multipage project, you will create multiple such projects. It is the single page wherein our script files will get injected by the build workflow (which is why you do not see scripts import here).

You can edit this file, but we will not write any html code here.

```
<div id="root"></div>
```

The above `<div>` is where we mount our entire react application. You can add css (`<link>`) or meta tag but no html.

Manifest.json → It is there because ‘create-react-app’ gives us a progressive web app out of the box. This file has some metadata about our application.

d) src

The files in src are the files we work in. This is our react application.

Index.js

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

The above line gets access to the root element in our DOM and it renders our react application with render() method. The App is the App component defined in the App.js.

App.css → Defines styling used in App.js file

index.css → Generic styling

serviceWorker.js → to register a service worker which is generated automatically, which is related to the progressive app we get out of the box.

App.test.js → Allows us to create unit tests for different units (for ex; components) in our application.

3.5. Understanding Component Basics

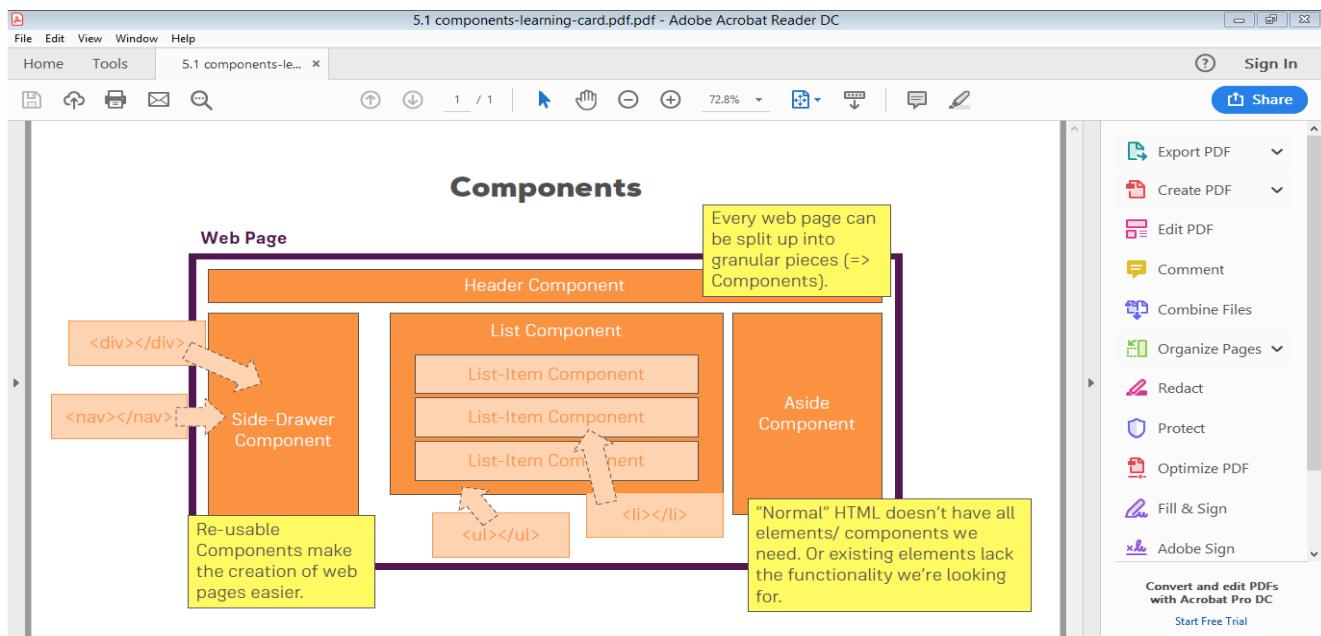
Components are custom HTML elements.

Typically in React, you render one root component (App.js here), and in this component all other components will be nested.

React will call render() method to render something to the screen.

Every react component has to return/render HTML code that can be rendered to the DOM/screen. However you can do other things also in a component.

3.5.1 components-learning-card.pdf



3.6. Understanding JSX

App.js

```
import React, { Component } from "react";
import "./App.css";

class App extends Component {
  render() {
    /* return (
      <div className="App">
        <h1> I will not React</h1>
      </div>
    ); */

    // The above html(JSX) code is converted into JavaScript by React.
    // The generated compiled code (compilation done by one of the build tools)
    will be as follows:
    // createElement() takes at least 3 arguments
    // The 1st arg is html element or our custom component
    // The 2nd argument is the configuration for the first argument
    // The 3rd argument is any amount of children separated by comma
    // Here h1 will be rendered as text, not html tag. This is the default
    behavior

    // return React.createElement("div", null, "h1", "I will not React");

    // If we want to render h1 as an html element then replace h1 onwards args to
    'React.createElement'
    // Also apply css ({className : 'App'}) in the configuration argument
    // As this code is cumbersome to write, we write the above JSX code
    return React.createElement(
      "div", { className: "App" },
      React.createElement("h1", null, "I will not React")
    );
  }
}

export default App;
```

3.7. JSX Restrictions

- (1) In JSX we cannot use the reserved keyword of JavaScript. For ex, we need to use 'className' in place of 'class' as class is a reserved keyword in JavaScript.

The JSX elements (`<div>`, `<h1>`) used in the previous steps are provided by **react library**. We are not using the real html tags. React is converting them behind the scenes. And react also defines the attributes that we can define on these elements.

(2) Our JSX expression must have one root element. It is the best practice to wrap everything in a root element (for ex; <div>).

Note: JSX is not HTML, but in most of the cases it works just like HTML.

3.8. Creating a Functional Component

React is all about components. React is a library that makes it easy to build component.

Create a new component:

(1) Add a new folder (Person) in src

(2) Create Person.js in Person folder

Most of the times, you should use a function to create a react component bcz in its simplest form, a component is a function which returns some JSX. Use ES6 syntax while creating react projects.

Person.js

```
import React from "react";
const person = () => {
    // We need to import 'React' in order to convert this code to JSX code
    (createElement())
    return <p> I 'm a person</p>;
};
export default person;
```

(3) Import Person component into App.js in order to use it

```
import React, { Component } from "react";
import "./App.css";
/* You can choose any name after import keyword, but it should start with uppercase.
Bcz in react JSX, all elements starting with lower case are reserved for
the native html element.
.js is optional as it is added automatically by build tool.
*/
import Person from "./Person/Person.js";

class App extends Component {
  render() {
    return (
      <div className="App">
        <h1> I will not React </h1> <Person />
      </div>
    );
  }
}
export default App;
```

3.9. Components & JSX Cheat Sheet

Components are the **core building block of React apps**. Actually, React really is just a library for creating components in its core.

A typical React app therefore could be depicted as a **component tree** - having one root component ("App") and then a potentially infinite amount of nested child components.

Each component needs to return/ render some **JSX** code - it defines which HTML code React should render to the real DOM in the end.

JSX is NOT HTML but it looks a lot like it. Differences can be seen when looking closely though (for example `className` in JSX vs `class` in "normal HTML"). JSX is just syntactic sugar for JavaScript, allowing you to write HTMLish code instead of nested `React.createElement(...)` calls.

When creating components, you have the choice between **two different ways**:

1. **Functional components** (also referred to as "presentational", "dumb" or "stateless" components - more about this later in the course) `=> const cmp = () => { return <div>some JSX</div> }` (using ES6 arrow functions as shown here is recommended but optional)
2. **class-based components** (also referred to as "containers", "smart" or "stateful" components) `=> class Cmp extends Component { render () { return <div>some JSX</div> } }`

We'll of course dive into the difference throughout this course, you can already note that you should use 1) as often as possible though. It's the best-practice.

3.10. Working with Components & Re-Using Them

We can reuse components by just using the component's name (`<Person />`) tag.

Make component configurable (next lecture).

3.11. Outputting Dynamic Content

Wrap dynamic content in {}, otherwise it will show as text.

Person.js

```
import React from "react";
const person = () => {
  // We need to import 'React' in order to convert this code to JSX code
  (createElement())
  return (
    <p>I'm a person and I am {Math.floor(Math.random() * 30)} years old</p>
  );
}
```

```
};

export default person;
```

3.12. Working with Props

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";
class App extends Component {
  render() {
    return (
      <div className="App">
        <h1> I will not React</h1>
        <Person name="Ram" age="20" />
        <Person name="Shyam" age="25" />
      </div>
    );
  }
}
export default App;
```

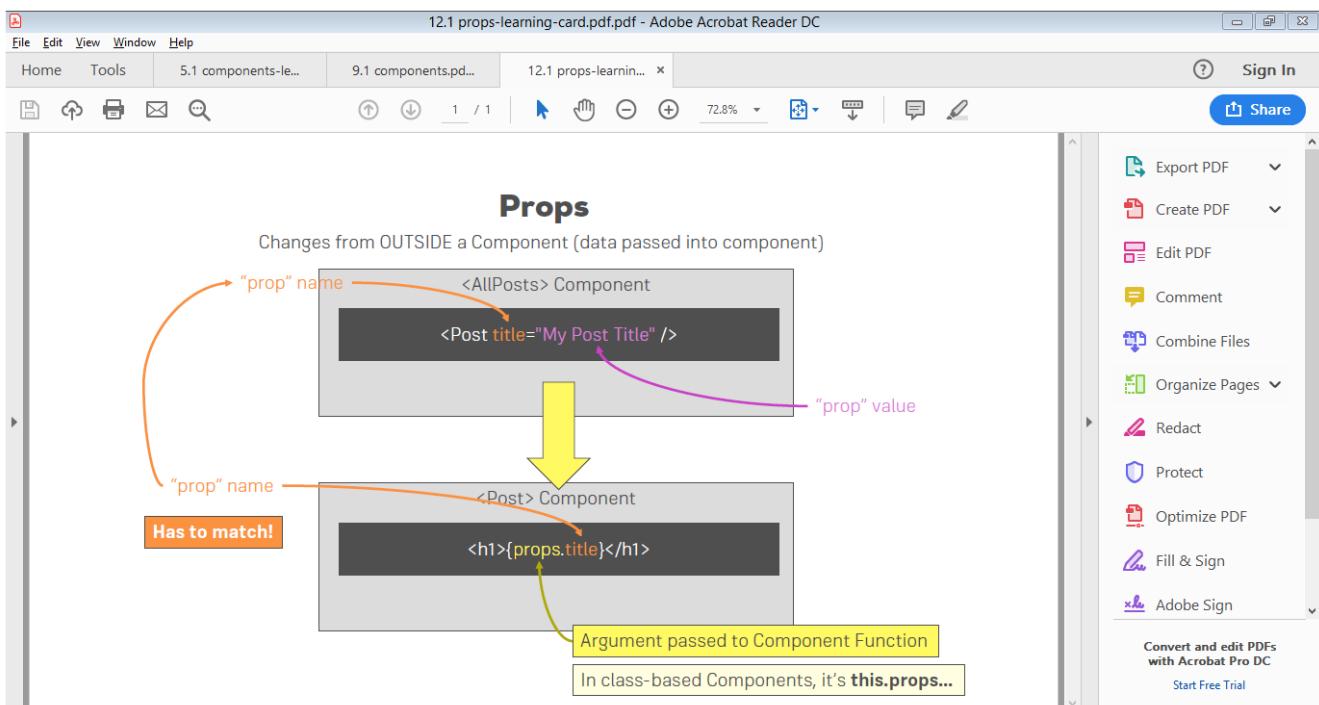
Person.js

```
import React from "react";
const person = props => {
  // We need to import 'React' in order to convert this code to JSX code
  (createElement())
  return (
    <p>
      I'm a person and My name is {props.name} and I am {props.age} years old
    </p>
  );
};
export default person;
```

When using class-based components, it's **this.props**

```
class Person extends Component {  
  render () {  
    return <p>My name is {this.props}</p>;  
  }  
}
```

3.12.1 props-learning-card.pdf



3.13. Understanding the Children Property

Output whatever we pass between opening and closing tags of our custom component.

{props.children} → Will fetch all the child elements including any complex HTML content

App.js

```
import React, { Component } from "react";  
import "./App.css";  
import Person from "./Person/Person.js";  
  
class App extends Component {
```

```

render() {
  return (
    <div className="App">
      <h1> I will not React</h1>
      <Person name="Ram" age="20" />
      <Person name="Shyam" age="25">
        {" "}
        My Hobby : TT{" "}
      </Person>
    </div>
  );
}
export default App;

```

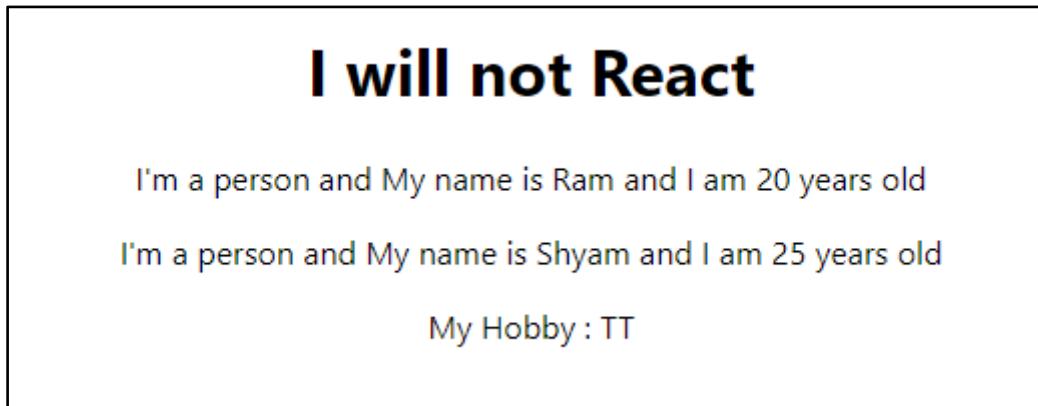
Person.js

```

import React from "react";
const person = props => {
  // We need to import 'React' in order to convert this code to JSX code
  (createElement())
  return (
    <div>
      <p>
        I'm a person and My name is {props.name} and I am {props.age} years old
      </p>
      <p> {props.children}</p>
    </div>
  );
};
export default person;

```

o/p:



3.14. Understanding & Using State

In the previous lecture we got info from outside using props. Sometimes you do not want to get info from outside, but you want to have it inside the component and change it from inside too.

For ex, create a button in App component that changes the name after switching it on. We will handle this in the next lecture.

First of all we need to define the names in App.js in non-hardcoded way.

There is one property (state) that you can define in any component that extends Component. You cannot do it in a functional component. State property is managed from inside the component.

You should use functional component as often as possible because you should use state with care. Having state in all your components and manipulating it from anywhere in your app makes your app unpredictable and hard to manage especially as it grows.

State (used to store component internal data) takes an object as a value.

This (keyword) refers to the current class due to ES6 syntax.

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { name: "Ram", age: "20" },
      { name: "Shyam", age: "25" },
      { name: "Mohan", age: "28" }
    ]
  };
  render() {
    return (
      <div className="App">
        <h1> I will not React</h1>
        <button>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}
        >
      </div>
    );
  }
}
```

```

        {" "}
        My Hobby : TT{" "}
    </Person>
    <Person
        name={this.state.persons[2].name}
        age={this.state.persons[2].age}
    />
    </div>
);
}
export default App;

```

State can be changed and if it changes, it will lead react to update our DOM. Next lecture.

3.15. Props & State

`props` and `state` are **CORE concepts** of React. Actually, only changes in `props` and/or `state` trigger React to re-render your components and potentially update the DOM in the browser (a detailed look at how React checks whether to really touch the real DOM is provided in section 6).

Props

`props` allow you to pass data from a parent (wrapping) component to a child (embedded) component.

Example:

AllPosts Component:

```

const posts = () => {
    return (
        <div>
            <Post title="My first Post" />
        </div>
    );
}

```

Here, `title` is the custom property (`prop`) set up on the custom `Post` component. We basically replicate the default HTML attribute behavior we already know (e.g. `<input type="text">` informs the browser about how to handle that input).

Post Component:

```
const post = (props) => {
```

```

        return (
            <div>
                <h1>{props.title}</h1>
            </div>
        );
    }
}

```

The `Post` component receives the `props` argument. You can of course name this argument whatever you want - it's your function definition, React doesn't care! But React will pass one argument to your component function => An object, which contains all properties you set up on `<Post ... />`.

`{props.title}` then dynamically outputs the `title` property of the `props` object - which is available since we set the `title` property inside `AllPosts` component (see above).

State

Whilst `props` allow you to pass data down the component tree (and hence trigger an UI update), state is used to change the component, well, state from within. Changes to state also trigger an UI update.

Example:

NewPost Component:

```

class NewPost extends Component { // state can only be accessed in class-based
components!
    state = {
        counter: 1
    };

    render () { // Needs to be implemented in class-based components! Needs to
return some JSX!
        return (
            <div>{this.state.counter}</div>
        );
    }
}

```

Here, the `NewPost` component contains `state`. Only class-based components can define and use `state`. You can of course pass the `state` down to functional components, but these then can't directly edit it.

`state` simply is a property of the component class, you have to call it `state` though - the name is not optional. You can then access it via `this.state` in your class JSX code (which you return in the required `render()` method).

Whenever `state` changes (taught over the next lectures), the component will re-render and reflect the new state. The difference to `props` is, that this happens within one and the same component - you don't receive new data (`props`) from outside!

3.16. Handling Events with Methods

In normal JavaScript event is: `onClick`, but in JSX it is `onClick`.

Note: `<button onClick = {this.switchNameHandler}>Switch name</button>`

Do not use `'onClick = {this.switchNameHandler()}'` → This will immediately execute it once react renders JSX to the DOM. We only want to pass a reference and we do this by using `this` and then referring to the property which holds the function.

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { name: "Ram", age: "20" },
      { name: "Shyam", age: "25" },
      { name: "Mohan", age: "28" }
    ]
  };

  // It is a convention to append the name with Handler.
  /* Handler tells that this is a method you are not actively calling,
   but you are assigning as an event handler
  */
  /*
    if you do not use this syntax, you will run into errors if you try to use 'this'.
    Bcz 'this' will not then refer to the class at runtime.
  */
  switchNameHandler = () => {
    console.log("Was clicked!!");
  };

  render() {
    return (
      <div className="App">
        <h1> I will not React</h1>
        <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 : {this.state.persons[1].hobby}
    </Person>
    <Person
      name={this.state.persons[2].name}
      age={this.state.persons[2].age}
    />
  </div>
);
}
export default App;

```

3.17. To Which Events Can You Listen

In the last lecture, we saw that you can react to the onClick event - but to which other events can you listen? You can find a list of supported events here: <https://reactjs.org/docs/events.html#supported-events>

Clipboard Events

Event names:

onCopy onCut onPaste

Properties:

DOMDataTransfer clipboardData

Composition Events

Event names:

onCompositionEnd onCompositionStart onCompositionUpdate

Properties:

string data

Keyboard Events

Event names:

```
onKeyDown onKeyPress onKeyUp
```

Properties:

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

Focus Events

Event names:

```
onFocus onBlur
```

These focus events work on all elements in the React DOM, not just form elements.

Properties:

```
DOMEventTarget relatedTarget
```

Form Events

Event names:

```
onChange onInput onInvalid onSubmit
```

For more information about the onChange event, see [Forms](#).

Mouse Events

Event names:

```
onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
onMouseMove onMouseOut onMouseOver onMouseUp
```

The `onMouseEnter` and `onMouseLeave` events propagate from the element being left to the one being entered instead of ordinary bubbling and do not have a capture phase.

Properties:

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEVENTTARGET relatedTarget
number screenX
number screenY
boolean shiftKey
```

Selection Events

Event names:

```
onSelect
```

Touch Events

Event names:

```
onTouchCancel onTouchEnd onTouchMove onTouchStart
```

Properties:

```
boolean altKey
DOMTOUCHLIST changedTouches
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTOUCHLIST targetTouches
DOMTOUCHLIST touches
```

UI Events

Event names:

```
onScroll
```

Properties:

```
number detail
```

```
DOMAbstractView view
```

Wheel Events

Event names:

```
onWheel
```

Properties:

```
number deltaMode  
number deltaX  
number deltaY  
number deltaZ
```

Media Events

Event names:

```
onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted  
onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay  
onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend  
onTimeUpdate onVolumeChange onWaiting
```

Image Events

Event names:

```
onLoad onError
```

Animation Events

Event names:

```
onAnimationStart onAnimationEnd onAnimationIteration
```

Properties:

```
string animationName  
string pseudoElement  
float elapsedTime
```

Transition Events

Event names:

```
onTransitionEnd
```

Properties:

```
string propertyName
string pseudoElement
float elapsedTime
```

Other Events

Event names:

onToggle

3.18. Manipulating the State

Change the state.

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { name: "Ram", age: "20" },
      { name: "Shyam", age: "25" },
      { name: "Mohan", age: "28" }
    ],
    otherState: "Default state"
  };

  switchNameHandler = () => {
    // console.log("Was clicked!!");
    // We should not mutate (change) the state directly like the below.
    // react will not recognize and pickup this change.
    // Do not do this: this.state.persons[0].name = "Ramavatar";

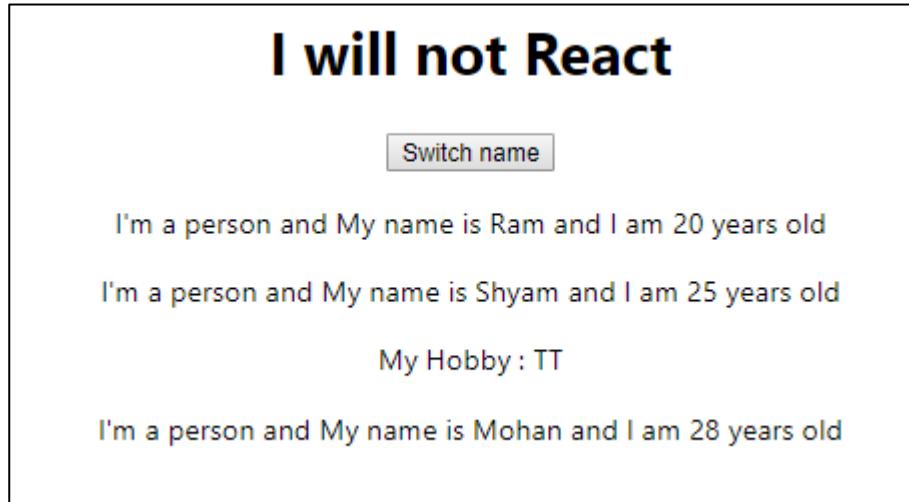
    // This will only change the state for 'persons'. 'otherState' property will be
    // unchanged
    // in other words, it will merge the old state with the new one
    this.setState({
      persons: [
        { name: "Ramavatar", age: "20" },
        { name: "Shyam", age: "25" },
        { name: "Mohan", age: "29" }
      ]
    });
  };
}
```

```

render() {
  return (
    <div className="App">
      <h1> I will not React</h1>
      <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 : TT{" "}
      </Person>
      <Person
        name={this.state.persons[2].name}
        age={this.state.persons[2].age}
      />
    </div>
  );
}
export default App;

```

O/P:



O/P After clicking button 'Switch name' :

I will not React

[Switch name](#)

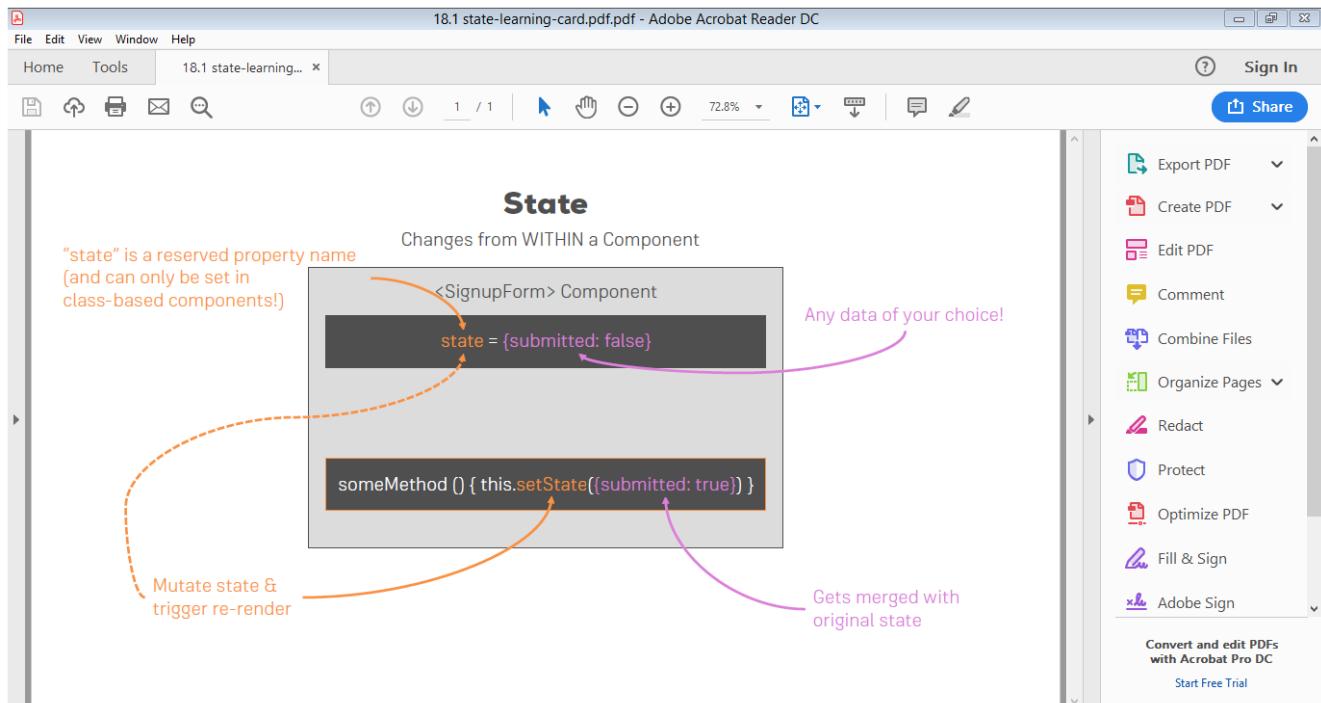
I'm a person and My name is Ramavatar and I am 20 years old

I'm a person and My name is Shyam and I am 25 years old

My Hobby : TT

I'm a person and My name is Mohan and I am 29 years old

3.18.1 state-learning-card.pdf



3.19. Functional (Stateless) vs class (Stateful) Components

props and state are the only 2 things that makes react update your DOM if something changes.

Components containing state is called **containers**.

Changing state from another component. Next lecture.

3.20. Passing Method References between Components

Let's call switchNameHandler not only when we click 'switch name' button but also when we click any of the paragraphs inside a person component.

For this, in person component we can add onClick, but we cannot call a method (here switchNameHandler) from a different file/class.

We can pass reference to this handler (switchNameHandler) as a property to our component.

Add this to App.js to the paragraph: click= {this. switchNameHandler} → we can take any name in place of click.

In Person.js → onClick = {props.click}

Person.js

```
import React from "react";
const person = props => {
  return (
    <div>
      <p onClick={props.click}>
        I'm a person and My name is {props.name} and I am {props.age} years old
      </p>
      <p> {props.children}</p>
    </div>
  );
};
export default person;
```

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { name: "Ram", age: "20" },
      { name: "Shyam", age: "25" },
      { name: "Mohan", age: "28" }
    ],
    otherState: "Default state"
  };

  switchNameHandler = () => {
    this.setState({
      persons: [
```

```

        { name: "Ramavatar", age: "20" },
        { name: "Shyam", age: "25" },
        { name: "Mohan", age: "29" }
    ]
});
};

render() {
    return (
        <div className="App">
            <h1> I will not React</h1>
            <button onClick={this.switchNameHandler}>Switch name</button>
            <Person
                name={this.state.persons[0].name}
                age={this.state.persons[0].age}
            />
            {/* Add a property 'click' and pass ref to handler click=
{this.switchNameHandler} */}
            <Person
                name={this.state.persons[1].name}
                age={this.state.persons[1].age}
                click={this.switchNameHandler}
            >
                My Hobby : TT
            </Person>
            <Person
                name={this.state.persons[2].name}
                age={this.state.persons[2].age}
            />
        </div>
    );
}
}

export default App;

```

We also want to pass a value to our function (switchNameHandler()). There are 2 ways to pass data to handler.

1st way: (onClick={this.switchNameHandler.bind(this, 'Ramavatar')})

'this' controls what 'this' inside the function will refer to.

App.js

```

import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

```

```

class App extends Component {
  state = {
    persons: [
      { name: "Ram", age: "20" },
      { name: "Shyam", age: "25" },
      { name: "Mohan", age: "28" }
    ],
    otherState: "Default state"
  };

  switchNameHandler = newName => {
    this.setState({
      persons: [
        { name: newName, age: "20" },
        { name: "Shyam", age: "25" },
        { name: "Mohan", age: "29" }
      ]
    });
  };

  render() {
    return (
      <div className="App">
        <h1> Hello React!</h1>
        <button onClick={this.switchNameHandler.bind(this, "Ramavatar")}>
          Switch name
        </button>
        <Person
          name={this.state.persons[0].name}
          age={this.state.persons[0].age}
        />
        {/* Add a property 'click' and pass ref to handler
        click={this.SwitchNameHandler} */}
        <Person
          name={this.state.persons[1].name}
          age={this.state.persons[1].age}
          click={this.switchNameHandler.bind(this, "Ram Krishna")}
        >
          My Hobby : TT
        </Person>
        <Person
          name={this.state.persons[2].name}
          age={this.state.persons[2].age}
        />
      </div>
    );
  }
}

```

```
}
```

```
export default App;
```

2nd way: (AVOID)

onClick={() => this.switchNameHandler('Ramavatar')} → This will not get executed immediately. Instead what we pass here is an anonymous function which will be executed on a click and which tends to return the result of the function (switchNameHandler()) getting executed.

This is not an efficient way. React can re-render certain things in your app too often. Avoid this syntax.

App.js

```
<button onClick={() => this.switchNameHandler("Ramavatar")}>
    Switch name
</button>
<Person
    name={this.state.persons[0].name}
    age={this.state.persons[0].age}
/>
```

3.21. Adding Two Way Binding

We want to change the name on our own.

Person.js

```
import React from "react";
const person = props => {
  return (
    <div>
      <p onClick={props.click}>
        I'm a person and My name is {props.name} and I am {props.age} years old
      </p>
      <p> {props.children}</p>
      <input type="text" onChange={props.changed} />
    </div>
  );
};
export default person;
```

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";
```

```

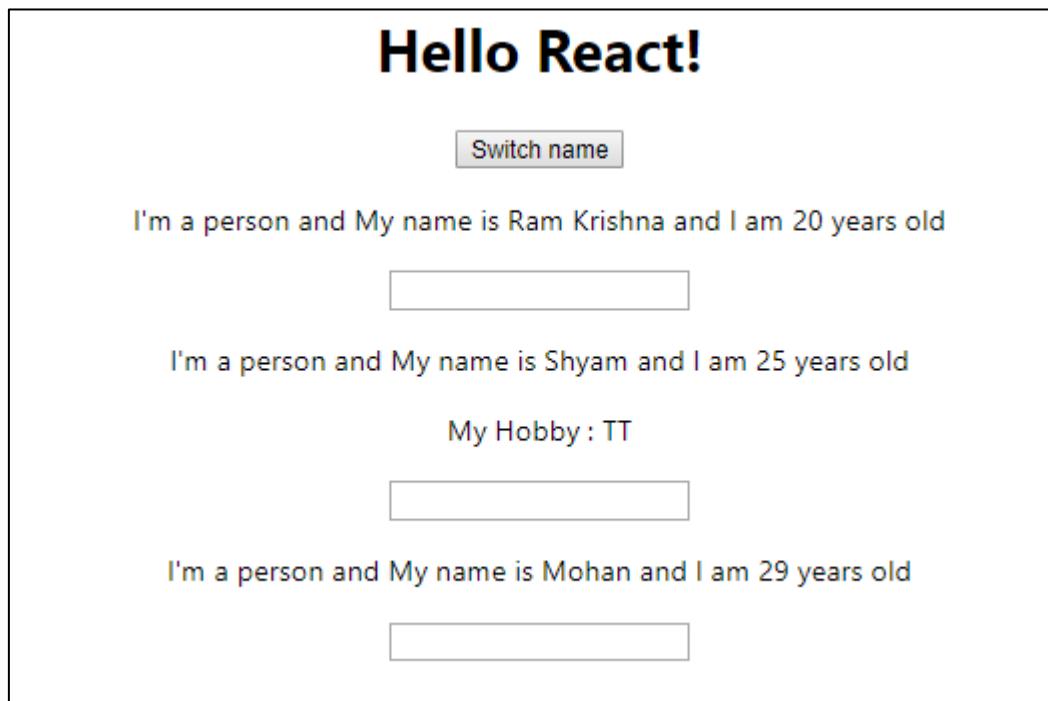
class App extends Component {
  state = {
    persons: [
      { name: "Ram", age: "20" },
      { name: "Shyam", age: "25" },
      { name: "Mohan", age: "28" }
    ],
    otherState: "Default state"
  };

  switchNameHandler = newName => {
    this.setState({
      persons: [
        { name: newName, age: "20" },
        { name: "Shyam", age: "25" },
        { name: "Mohan", age: "29" }
      ]
    });
  };
  // The event object is automatically passed by JavaScript.
  nameChangedHandler = event => {
    this.setState({
      persons: [
        { name: "Ramavatar", age: "20" },
        { name: event.target.value, age: "25" },
        { name: "Mohan", age: "28" }
      ]
    });
  };
}

render() {
  return (
    <div className="App">
      <h1> Hello React!</h1>
      <button onClick={() => this.switchNameHandler("Ramavatar")}>
        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, "Ram Krishna")}
        changed={this.nameChangedHandler}
      >
    
```

```
    My Hobby : TT
  </Person>
  <Person
    name={this.state.persons[2].name}
    age={this.state.persons[2].age}
  />
  </div>
);
}
export default App;
```

o/p:



Hello React!

[Switch name](#)

I'm a person and My name is Ramavatar and I am 20 years old

aaaaaaaaaaaaaa

I'm a person and My name is bbbbbbbbbbbbbb and I am 25 years old

My Hobby : TT

bbbbbbbbbbbbbbbb

I'm a person and My name is Mohan and I am 28 years old

cccccccccccccccccccccccc

2-way binding:

Person.js

```
import React from "react";
const person = props => {
  return (
    <div>
      <p onClick={props.click}>
        I'm a person and My name is {props.name} and I am {props.age} years old
      </p>
      <p> {props.children}</p>
      {/* onChange will change the value in paragraph.
      'value' will take the changed value and display in the input box.
      This is 2-way binding. */}
    </div>
  );
};
export default person;
```

o/p:

Hello React!

I'm a person and My name is Ram and I am 20 years old

I'm a person and My name is Shyam and I am 25 years old

My Hobby : TT

I'm a person and My name is Mohan and I am 28 years old

O/P after typing some value in 2nd textbox:

Hello React!

I'm a person and My name is and I am 20 years old

I'm a person and My name is and I am 25 years old

My Hobby : TT

I'm a person and My name is Mohan and I am 28 years old

3.22. Adding Styling with Stylesheets

Add Person.css to person component.

Also import Person.css in to Person.js. By doing this, you make your webpack aware of this and it will add this css into html file dynamically.

Person.css

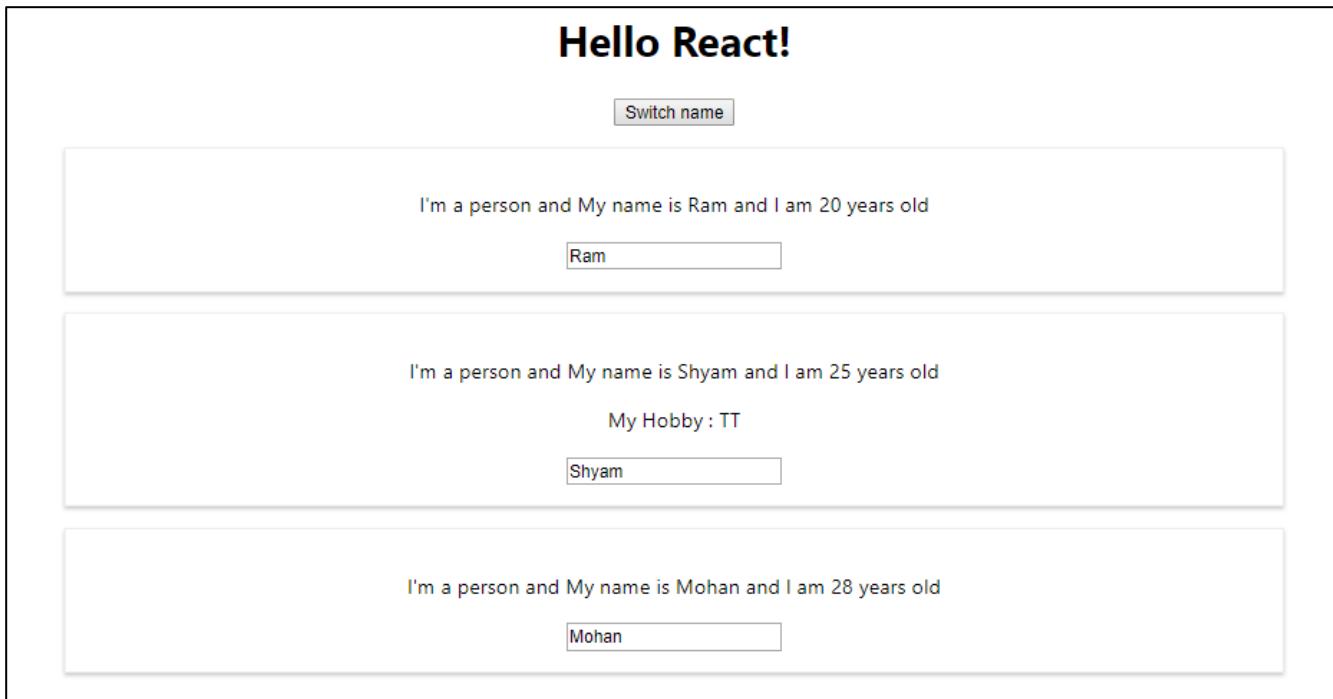
```
/* if we do not write .Person, this stylesheet will become global*/
.Person {
  width: 60%;
  margin: 16px auto;
  border: 1px solid #eee;
  box-shadow: 0 2px 3px #ccc;
  padding: 16px;
  text-align: center;
}
```

Person.js

```
import React from "react";
import "./Person.css";

const person = props => {
  // We need to import 'React' in order to convert this code to JSX code
  (createElement())
  return (
    <div className="Person">
      <p onClick={props.click}>
        I'm a person and My name is {props.name} and I am {props.age} years old
      </p>
      <p> {props.children}</p>
      {/* onChange will change the value in paragraph.
      'value' will take the changed value and display in the input box.
      This is 2-way binding.
    */}
      <input type="text" onChange={props.changed} value={props.name} />
    </div>
  );
};
export default person;
```

o/p:



3.23. Working with Inline Styles

Create a JavaScript const for styling.

Inline style does not allow us to leverage full power of css.

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { name: "Ram", age: "20" },
      { name: "Shyam", age: "25" },
      { name: "Mohan", age: "28" }
    ],
    otherState: "Default state"
  };

  switchNameHandler = newName => {
    this.setState({
      persons: [
        { name: newName, age: "20" },
        { name: "Shyam", age: "25" },
        { name: "Mohan", age: "28" }
      ]
    });
}
```

```

        { name: "Mohan", age: "29" }
    ]
});
};

// The event object is automatically passed by JavaScript.
nameChangedHandler = event => {
  this.setState({
    persons: [
      { name: "Ramavatar", age: "20" },
      { name: event.target.value, age: "25" },
      { name: "Mohan", age: "28" }
    ]
  });
};

render() {
  const style = {
    backgroundColor: "white",
    font: "inherit",
    border: "1px solid blue",
    padding: "8px",
    cursor: "pointer"
  };

  return (
    <div className="App">
      <h1> Hello React!</h1>
      <button
        style={style}
        onClick={() => this.switchNameHandler("Ramavatar")}
      >
        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, "Ram Krishna")}
        changed={this.nameChangedHandler}
      >
        My Hobby : TT
      </Person>
      <Person
        name={this.state.persons[2].name}
        age={this.state.persons[2].age}
      />
    </div>
  );
}

switchNameHandler(name) {
  this.setState({ persons: [ { name, age: "25" } ] });
}

nameChangedHandler(event) {
  this.setState({ persons: [ { name: event.target.value, age: "25" } ] });
}

```

```
        />
      </div>
    );
}
export default App;
```

3.25. Useful Resources & 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>

4. Working with Lists and Conditionals

4.1. Module Introduction

4.2. Rendering Content Conditionally

Wrap/hide the other persons after click on 'switch name'.

Steps:

- 1) Put all the <Person> tags inside a <div>.
- 2) Create handler (togglePersonsHandler).
- 3) Add a property to state object
- 4) Enclose the < div> into {} and use ternary operator as you can write JavaScript expressions inside {}

Note: We can inject JavaScript expression in JSX inside {}.

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { name: "Ram", age: "20" },
      { name: "Shyam", age: "25" },
      { name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
  };

  switchNameHandler = newName => {
    this.setState({
      persons: [
        { name: newName, age: "20" },
        { name: "Shyam", age: "25" },
        { name: "Mohan", age: "29" }
      ]
    });
  };
  // The event object is automatically passed by JavaScript.
  nameChangedHandler = event => {
    this.setState({
      persons: [
        { name: "Shyam", age: "25" }
      ]
    });
  };
}

export default App;
```

```

        { name: "Ramavatar", age: "20" },
        { name: event.target.value, age: "25" },
        { name: "Mohan", age: "28" }
    ]
});
};

togglePersonsHandler = () => {
    const doesState = this.state.showPersons;
    // This will merge 'showPersons' attribute with state object
    this.setState({ showPersons: !doesState });
};

render() {
    const style = {
        backgroundColor: "white",
        font: "inherit",
        border: "1px solid blue",
        padding: "8px",
        cursor: "pointer"
    };
    return (
        <div className="App">
            <h1> Hello React!</h1>
            <button style={style} onClick={this.togglePersonsHandler}>
                Switch name
            </button>
            {this.state.showPersons ? (
                <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, "Ram Krishna")}
                        changed={this.nameChangedHandler}
                    >
                        My Hobby : TT
                    </Person>
                    <Person
                        name={this.state.persons[2].name}
                        age={this.state.persons[2].age}
                    />
                </div>
            ) : null}
        </div>
    );
}

```

```
    );
}
}

export default App;
```

o/p:

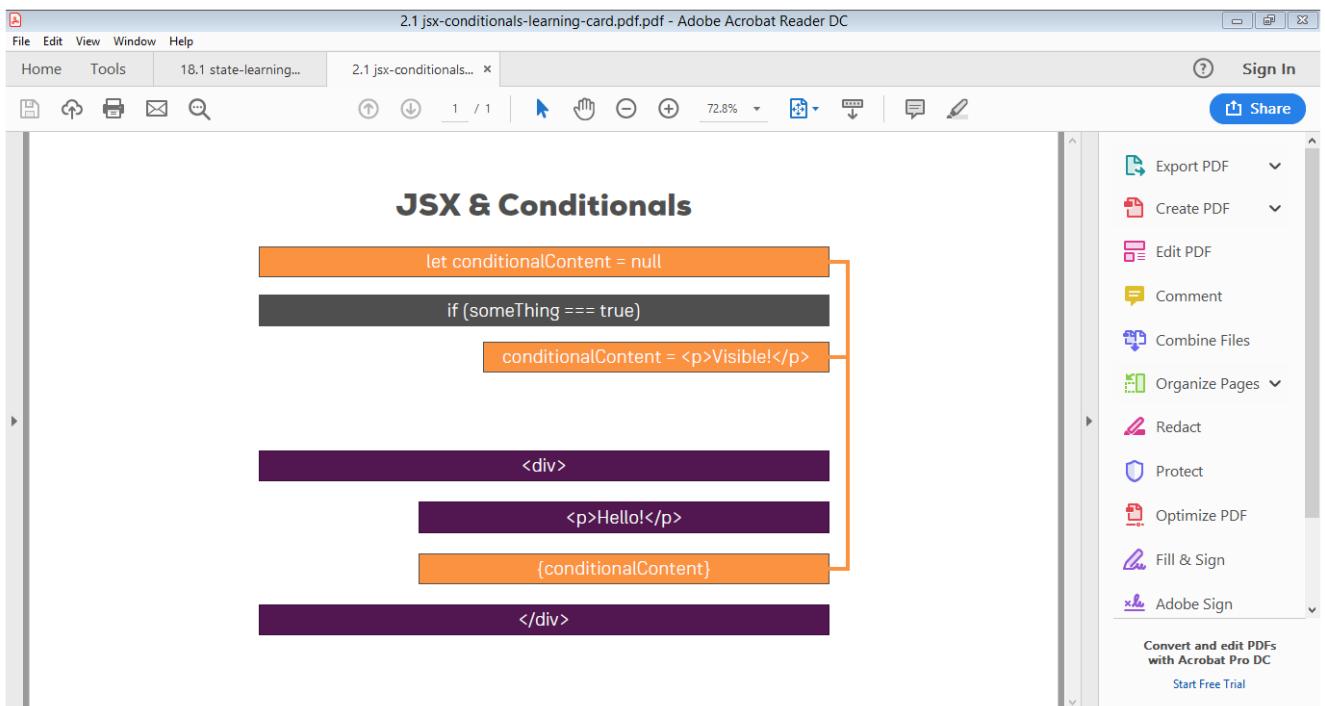


After clicking 'Switch name' button:

A screenshot of a web browser showing the state of the React application after the 'Switch name' button has been clicked. The page contains three distinct horizontal cards, each with a different person's information. The first card displays 'I'm a person and My name is Ram and I am 20 years old' above a text input field containing the value 'Ram'. The second card displays 'I'm a person and My name is Shyam and I am 25 years old' above a text input field containing the value 'Shyam'. The third card displays 'I'm a person and My name is Mohan and I am 28 years old' above a text input field containing the value 'Mohan'. Each card is enclosed in its own light gray box, and the entire set is contained within a larger white box.

Writing conditionals like the above can lead to confusing code if you have lot of nested checks. An alternative to this is discussed in the next lecture.

[4.2.1 jsx-conditionals-learning-card.pdf](#)



4.3. Handling Dynamic Content the JavaScript Way

Everything inside the render() method gets executed every time react re-renders component. So we can add some code before we return something.

App.js

```

import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { name: "Ram", age: "20" },
      { name: "Shyam", age: "25" },
      { name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
  };

  switchNameHandler = newName => {
    this.setState({
      persons: [
        { name: newName, age: "20" },
        { name: "Shyam", age: "25" },
        { name: "Mohan", age: "29" }
      ]
    });
  }
}

export default App;

```

```

        ]
    });
};

// The event object is automatically passed by javascript.
nameChangedHandler = event => {
    this.setState({
        persons: [
            { name: "Ramavatar", age: "20" },
            { name: event.target.value, age: "25" },
            { name: "Mohan", age: "28" }
        ]
    });
};

togglePersonsHandler = () => {
    const doesState = this.state.showPersons;
    // This will merge 'showPersons' attribute with state object
    this.setState({ showPersons: !doesState });
};

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, "Ram Krishna")}
                    changed={this.nameChangedHandler}
                >
                    My Hobby : TT
                </Person>
                <Person
                    name={this.state.persons[2].name}

```

```

        age={this.state.persons[2].age}
      />
    </div>
  );
}

return (
  <div className="App">
    <h1> Hello React!</h1>
    <button style={style} onClick={this.togglePersonsHandler}>
      Toggle persons
    </button>
    {persons}
  </div>
);
}
export default App;

```

4.4. Outputting Lists (Intro)

Apply list on the persons array in state object.

4.5. Outputting Lists

In vue we have a directive v-for, In angular we have ngFor to iterate over list in <person>. But in react we do not have that because everything is JavaScript in react.

{this.state.persons} → is not a valid JSX, so we need to convert it to valid JSX using map() function.

Map() maps every element in a given array into something else. It does this by executing a method on every element in a given array. Map() takes element of the array as input.

```

{this.state.persons.map(person => {
  return <Person name={person.name} age={person.age}/>
})
}
```

App.js

```

import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [

```

```

        { name: "Ram", age: "20" },
        { name: "Shyam", age: "25" },
        { name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
};

switchNameHandler = newName => {
    this.setState({
        persons: [
            { name: newName, age: "20" },
            { name: "Shyam", age: "25" },
            { name: "Mohan", age: "29" }
        ]
    });
};

// The event object is automatically passed by javascript.
nameChangedHandler = event => {
    this.setState({
        persons: [
            { name: "Ramavatar", age: "20" },
            { name: event.target.value, age: "25" },
            { name: "Mohan", age: "28" }
        ]
    });
};

togglePersonsHandler = () => {
    const doesState = this.state.showPersons;
    // This will merge 'showPersons' attribute with state object
    this.setState({ showPersons: !doesState });
};

render() {
    const style = {
        backgroundColor: "white",
        font: "inherit",
        border: "1px solid blue",
        padding: "8px",
        cursor: "pointer"
    };

    let persons = null;
    if (this.state.showPersons) {
        persons = (
            <div>

```

```

    /* Mapping a JavaScript array to an array having JSX object inside */
    {this.state.persons.map(person => {
      return <Person name={person.name} age={person.age} />;
    })}
  </div>
);
}

return (
  <div className="App">
    <h1> Hello React!</h1>
    <button style={style} onClick={this.togglePersonsHandler}>
      Toggle persons
    </button>
    {persons}
  </div>
);
}
export default App;

```

o/p:

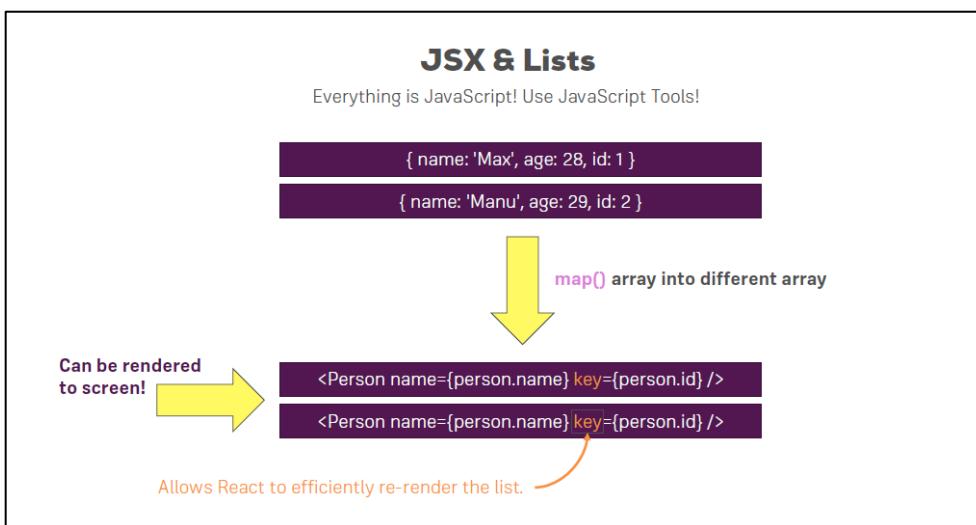


O/P after clicking 'switch name' button:

The screenshot shows a React application with three cards. Each card contains a text description and a text input field. The browser's developer tools are open, displaying two warning messages:

- Warning: Each child in a list should have a unique "key" prop. index.js:1446
- Warning: Failed prop type: You provided a 'value' prop to a form field without an 'onChange' handler. This will render a read-only field. If the field should be mutable use 'defaultValue'. Otherwise, set either 'onChange' or 'readOnly'. index.js:1446

4.5.1 jsx-lists-learning-card.pdf



4.6. Lists & State

Key error in the above screenshot.

How to react to changes to array?

How to change array by using event listener?

To delete a person from array of persons:

- 1) Create a deletePersonHandler
- 2) Add 'click' property to Person.

App.js

```

import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

```

```

class App extends Component {
  state = {
    persons: [
      { name: "Ram", age: "20" },
      { name: "Shyam", age: "25" },
      { name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
  };

  switchNameHandler = newName => {
    this.setState({
      persons: [
        { name: newName, age: "20" },
        { name: "Shyam", age: "25" },
        { name: "Mohan", age: "29" }
      ]
    });
  };
  // The event object is automatically passed by javascript.
  nameChangedHandler = event => {
    this.setState({
      persons: [
        { name: "Ramavatar", age: "20" },
        { name: event.target.value, age: "25" },
        { name: "Mohan", age: "28" }
      ]
    });
  };
  togglePersonsHandler = () => {
    const doesState = this.state.showPersons;
    // This will merge 'showPersons' attribute with state object
    this.setState({ showPersons: !doesState });
  };

  deletePersonHandler = personIndex => {
    const persons = this.state.persons;
    /* This will create a new version of persons array and will start splicing at
    personIndex and splice one element. This simply removes one element from the array.
     */
    persons.splice(personIndex, 1);
    this.setState({ persons: persons });
  };
}

```

```

render() {
  const style = {
    backgroundColor: "white",
    font: "inherit",
    border: "1px solid blue",
    padding: "8px",
    cursor: "pointer"
  };

  let persons = null;
  if (this.state.showPersons) {
    persons = (
      <div>
        {this.state.persons.map((person, index) => {
          return (
            <Person
              click={() => this.deletePersonHandler(index)}
              name={person.name}
              age={person.age}
            />
          );
        })}
      </div>
    );
  }
  return (
    <div className="App">
      <h1> Hello React!</h1>
      <button style={style} onClick={this.togglePersonsHandler}>
        Toggle persons
      </button>
      {persons}
    </div>
  );
}
}

export default App;

```

o/p:

The screenshot shows a React application titled "Hello React!" with three cards. Each card contains a name and an input field. The first card has "Ram" in its input field. The developer tools console shows two warning messages:

- Warning: Each child in a list should have a unique "key" prop. index.js:1446
- Warning: Failed prop type: You provided a 'value' prop to a form field without an 'onChange' handler. This will render a read-only field. If the field should be mutable use 'defaultValue'. Otherwise, set either 'onChange' or 'readOnly'. index.js:1446

O/P after clicking 'Ram' div:

The screenshot shows the same React application after clicking the "Ram" div. The first card's input field is now empty. The developer tools console shows the same two warning messages.

4.7. Updating State Immutable

```
deletePersonHandler = personIndex => {
  const persons = this.state.persons;
  /* This will create a new version of persons array and will start splicing at
  personIndex and splice one element. This simply removes one element from the array.
  */
  persons.splice(personIndex, 1);
  this.setState({ persons });
};
```

deletePersonHandler → The flaw in this approach in deleting the elements is that , JavaScript objects and arrays are reference types. 'persons' is just a pointer to the original 'persons' state managed by react. Splice mutate this original data and whilst it works without throwing error, it may lead to unpredictable apps. A good practice is to create a copy of persons array before manipulating it. Simple way to do it to call slice() method:

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

An alternative to the above approach would be:

```
const persons = [...this.state.persons]; //Modern Approach (ES6)
```

Note: You should always update state in an immutable way. Create a copy, change that and then update the state with setState().

4.8. Lists & Keys

The key error in the previous screenshots. Let's resolve it.

Key is a property react expects to find on an element no matter whether it is a custom component or a default HTML element which you render through a list (by mapping an array into JSX element). Key property should be added while rendering a list of data. Key property helps react update the list efficiently. Consider a case we have in the previous section where we are deleting an element from a list. It works, but behind the scenes, react needs to find out what it needs to adjust in the overall DOM. React has Virtual DOM where it compares what it would render now if it were to call the render method now or if it did actually adjust the real DOM with the result of the render method to the previous DOM it rendered. It does the comparison between the future vs the past. For a list it needs to find out which element needs to be changed. [By default it will re-render the whole list](#). For long list this is inefficient. So we should assign a key property to allow react to keep track of individual elements so that it has a clear property it can compare with different elements to find out which elements changed and which did not. So that it only re-renders the elements which did change and not the whole list. Key has to be something unique.

Key warning will be gone after this change.

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { id: "dudu", name: "Ram", age: "20" },
      { id: "wjom", name: "Shyam", age: "25" },
      { id: "ywey", name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
  };

  // The event object is automatically passed by javascript.
  nameChangedHandler = event => {
    this.setState({
      persons: [
```

```

        { name: "Ramavatar", age: "20" },
        { name: event.target.value, age: "25" },
        { name: "Mohan", age: "28" }
    ]
});
};

togglePersonsHandler = () => {
  const doesState = this.state.showPersons;
  // This will merge 'showPersons' attribute with state object
  this.setState({ showPersons: !doesState });
};

deletePersonHandler = personIndex => {
  //const persons = this.state.persons.slice(); //--> Approach 1
  const persons = [...this.state.persons]; // --> Approach 2 (ES6)
  persons.splice(personIndex, 1);
  this.setState({ persons: persons });
};

render() {
  const style = {
    backgroundColor: "white",
    font: "inherit",
    border: "1px solid blue",
    padding: "8px",
    cursor: "pointer"
  };

  let persons = null;
  if (this.state.showPersons) {
    persons = (
      <div>
        {/* */}
        {this.state.persons.map((person, index) => {
          return (
            <Person
              click={() => this.deletePersonHandler(index)}
              name={person.name}
              age={person.age}
              // Adding key to help react to find the change in DOM
              key={person.id}
            />
          );
        })}
      </div>
    );
  }
}

```

```

        }

        return (
            <div className="App">
                <h1> Hello React!</h1>
                <button style={style} onClick={this.togglePersonsHandler}>
                    Toggle persons
                </button>
                {persons}
            </div>
        );
    }
}

export default App;

```

o/p:

The screenshot shows a browser window with the title "Hello React!". Inside, there are three input fields, each with a label above it: "I'm a person and My name is Ram and I am 20 years old", "I'm a person and My name is Shyam and I am 25 years old", and "I'm a person and My name is Mohan and I am 28 years old". Each input field contains the name "Ram", "Shyam", and "Mohan" respectively. To the right of the browser window is the browser's developer tools interface, specifically the "Console" tab. It displays a warning message: "Warning: Failed prop type: You provided a 'value' prop to a form field without an 'onChange' handler. This will render a read-only field. If the field should be mutable use 'defaultValue'. Otherwise, set either 'onChange' or 'readOnly'." The message also includes file and line numbers: index.js:1446, Person.js:14, App.js:56, App.js:52, App.js:70, and App.js:7.

4.9. Flexible Lists

Add `changed={(event)=> this.nameChangedHandler(event, person.id)}` to `<Person>`. This anonymous function will get executed on `onChange` event.

`FindIndex()` → Find element in an array but then get the index of the element. It takes a function as an argument just like `map()` method. It executes this function on each element of the array. Here we are returning true or false based on whether we are looking for this person or not.

// Not a good practice to mutate state directly. A person in persons array is a JavaScript object.
 JavaScript objects are reference types. So we should not mutate them directly bcz we only get a pointer when we reach put to person in the next statement. Hence we would mutate the original object to which a pointer points.

```
const person = this.state.persons[personIndex];
```

So better approach is, to create a new JavaScript object and use the spread operator in front of the object being fetched.

```
const person = {  
  ...this.state.persons[personIndex]  
};
```

Alternative approach,

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

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { id: "dudu", name: "Ram", age: "20" },
      { id: "wjow", name: "Shyam", age: "25" },
      { id: "ywey", name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
  };

  // The event object is automatically passed by javascript.
  nameChangedHandler = (event, id) => {

    // We want to update the state of the person but only for one into which input
    // field we typed
    const personIndex = this.state.persons.findIndex(p => {
      return p.id === id;
    });

    // Not a good practice to mutate state directly. A person in persons array is a
    // javascript object. JavaScript objects are reference types. So we should not mutate
    // them directly bcz we only get a pointer when we reach put to person in the next
    // statement. Hence we would mutate the original object to which a pointer points.
    //const person = this.state.persons[personIndex];

    // Better approach (ES6)
    const person = {
      ...this.state.persons[personIndex]
    };
  }
}
```

```

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

// Manipulating the copied object, not the original one
person.name = event.target.value;

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

this.setState({ persons: persons });
};

deletePersonHandler = (personIndex) => {
  // const persons = this.state.persons.slice();
  const persons = [...this.state.persons];
  persons.splice(personIndex, 1);
  this.setState({ persons: persons });
}

togglePersonsHandler = () => {
  const doesShow = this.state.showPersons;
  this.setState({ showPersons: !doesShow });
}

render() {
  const style = {
    backgroundColor: "white",
    font: "inherit",
    border: "1px solid blue",
    padding: "8px",
    cursor: "pointer"
  };
  let persons = null;
  if (this.state.showPersons) {
    persons = (
      <div>
        {this.state.persons.map((person, index) => {
          return (
            <Person
              click={() => this.deletePersonHandler(index)}
              name={person.name}
              age={person.age}
              key={person.id}
              changed={(event) => this.nameChangedHandler(event, person.id)}
            />
          );
        })}
      </div>
    );
  }
}

```

```

}
return (
  <div className="App">
    <h1>Hi, I'm a React App</h1>
    <p>This is really working!</p>
    <button
      style={style}
      onClick={this.togglePersonsHandler}>Toggle Persons</button>
    {persons}
  </div>
);
// return React.createElement('div', {className: 'App'}, React.createElement('h1', null, 'Does this work now?'));
}
}
export default App;

```

o/p:



O/P after clicking 'Toggle Persons'. We no longer see the warning message.

5. Styling React Components & Elements

5.1. Module Introduction

5.2. Outlining the Problem Set

Dynamically adjust the button style.

5.3. Setting Styles Dynamically

Green background, white text for the ‘Toggle Persons’ button. Change the background color to red after clicking the ‘Toggle Persons’.

You can manipulate style with any JavaScript code. And at the end, you bind the style in return() to apply that.

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { id: "dudu", name: "Ram", age: "20" },
      { id: "wjow", name: "Shyam", age: "25" },
      { id: "ywey", name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
  };

  // The event object is automatically passed by javascript.
  nameChangedHandler = (event, id) => {
    // We want to update the state of the person but only for one into which input
    // field we typed
    const personIndex = this.state.persons.findIndex(p => {
      return p.id === id;
    });

    // Not a good practice to mutate state directly. A person in persons array is a
    // javascript object. Javascript objects are reference types. So we should not mutate
    // them directly bcz we only get a pointer when we reach put to person in the next
    // statement. Hence we would mutate the original object to which a pointer points.
    //const person = this.state.persons[personIndex];

    // Better approach (ES6)
```

```

const person = {
  ...this.state.persons[personIndex]
};

// Alternative approach
//const person = Object.assign({}, this.state.persons[personIndex]);
// Manipulating the copied object, not the original one
person.name = event.target.value;
const persons = [...this.state.persons];
persons[personIndex] = person;
this.setState({ persons });
};

togglePersonsHandler = () => {
  const doesState = this.state.showPersons;
  // This will merge 'showPersons' attribute with state object
  this.setState({ showPersons: !doesState });
};

deletePersonHandler = personIndex => {
  //const persons = this.state.persons.slice(); //--> Approach 1
  const persons = [...this.state.persons]; // --> Approach 2 (ES6)
  persons.splice(personIndex, 1);
  this.setState({ persons });
};

render() {
  const style = {
    backgroundColor: "green",
    color: "white",
    font: "inherit",
    border: "1px solid blue",
    padding: "8px",
    cursor: "pointer"
  };
  let persons = null;
  if (this.state.showPersons) {
    persons = (
      <div>
        {this.state.persons.map((person, index) => {
          return (
            <Person
              click={() => this.deletePersonHandler(index)}
              name={person.name}
              age={person.age}
              key={person.id}
              changed={event => this.nameChangedHandler(event, person.id)}
            />
          );
        })}
    );
  }
}

```

```

        })}
    </div>
);
style.backgroundColor = "red";
}
return (
<div className="App">
<h1> Hello React!</h1>
<button style={style} onClick={this.togglePersonsHandler}>
    Toggle persons
</button>
{persons}
</div>
);
}
export default App;

```

O/P:



O/P after clicking 'Toggle Persons':

The image shows the state of the application after the "Toggle persons" button has been clicked. The interface remains the same, but now there are three separate, semi-transparent rectangular boxes stacked vertically. Each box contains text: the top one says "I'm a person and My name is Ram and I am 20 years old" with "Ram" highlighted in red; the middle one says "I'm a person and My name is Shyam and I am 25 years old" with "Shyam" highlighted in red; and the bottom one says "I'm a person and My name is Mohan and I am 28 years old" with "Mohan" highlighted in red. The "Toggle persons" button is now red with white text.

5.4. Setting Class Names Dynamically

Change the class of `<p> This is really working!</p>` dynamically based on the no of elements in the persons array.

2 or less elements → Turn it RED.

1 or less → RED and BOLD

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { id: "dudu", name: "Ram", age: "20" },
      { id: "wjow", name: "Shyam", age: "25" },
      { id: "ywey", name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
  };

  // The event object is automatically passed by javascript.
  nameChangedHandler = (event, id) => {
    // We want to update the state of the person but only for one into which input
    // field we typed
    const personIndex = this.state.persons.findIndex(p => {
      return p.id === id;
    });

    // Not a good practice to mutate state directly. A person in persons array is a
    // javascript object. Javascript objects are reference types. So we should not mutate
    // them directly bcz we only get a pointer when we reach put to person in the next
    // statement. Hence we would mutate the original object to which a pointer points.
    //const person = this.state.persons[personIndex];

    // Better approach (ES6)
    const person = {
      ...this.state.persons[personIndex]
    };

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

```

// Manipulating the copied object, not the original one
person.name = event.target.value;

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

this.setState({ persons: persons });
};

togglePersonsHandler = () => {
  const doesState = this.state.showPersons;
  // This will merge 'showPersons' attribute with state object
  this.setState({ showPersons: !doesState });
};

deletePersonHandler = personIndex => {
  //const persons = this.state.persons.slice(); //--> Approach 1
  const persons = [...this.state.persons]; // --> Approach 2 (ES6)
  persons.splice(personIndex, 1);
  this.setState({ persons: persons });
};

render() {
  const style = {
    backgroundColor: "green",
    color: "white",
    font: "inherit",
    border: "1px solid blue",
    padding: "8px",
    cursor: "pointer"
  };

  let persons = null;
  if (this.state.showPersons) {
    persons = (
      <div>
        {this.state.persons.map((person, index) => {
          return (
            <Person
              click={() => this.deletePersonHandler(index)}
              name={person.name}
              age={person.age}
              key={person.id}
              changed={event => this.nameChangedHandler(event, person.id)}
            />
          );
        })}
    );
  }
}

```

```

        </div>
    );
    style.backgroundColor = "red";
}

// let classes = ['red', 'bold'].join() // "red bold"
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']
}

return (
    <div className="App">
        <h1> Hello React!</h1>
        <p className={classes.join(" ")}> This is really working!</p>
        <button style={style} onClick={this.togglePersonsHandler}>
            Toggle persons
        </button>
        {persons}
    </div>
);
}
export default App;

```

App.css

```

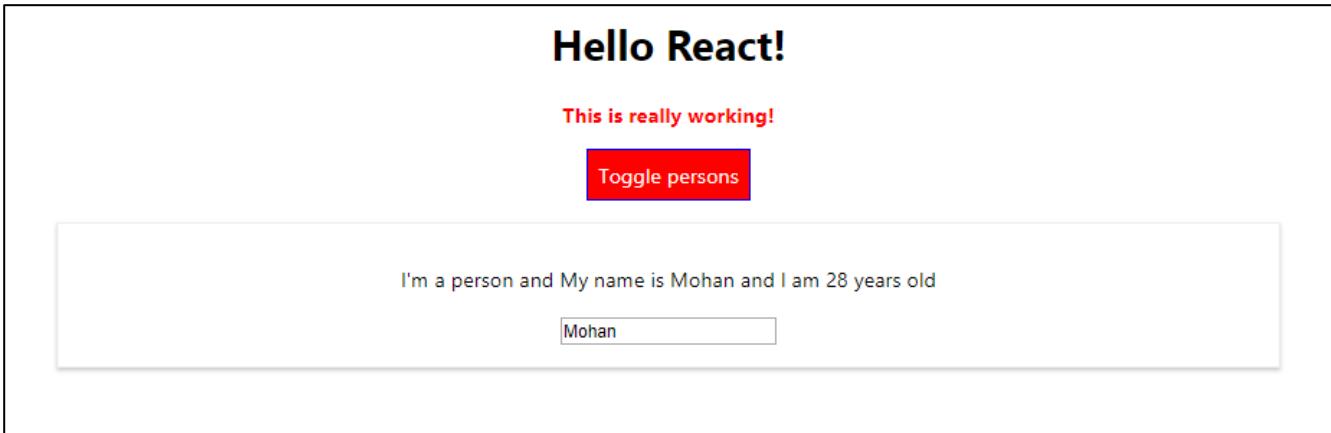
.App {
    text-align: center;
}

.red {
    color: red;
}

.bold {
    font-weight: bold;
}

```

o/p after deleting 2 elements:



5.5. Adding and Using Radium

'hover' is a pseudo selector, i.e. based on some other selector indicated by ':'.

D:\workspace\my-react-app> *npm install --save radium*

--save → to save an entry in package.json so that we also fix the version number and can easily share our project.

Radium is a popular package for react which [allows us to use inline styles with pseudo-selectors and media queries](#).

In order to use Radium, we need to import it into file where we want to use it.

Import Radium from 'radium';

Also do the following:

```
export default Radium(App);
```

This is called higher order component. It is just a component wrapping your component, injecting some extra functionality. This can be used with both (component created with class and functional component).

App.js

```
import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";
import Radium from "radium";

class App extends Component {
  state = {
    persons: [
      { id: "dudu", name: "Ram", age: "20" },
      { id: "wjow", name: "Shyam", age: "25" },
      { id: "ywey", name: "Mohan", age: "28" }
    ],
  }
}
```

```

        otherState: "Default state",
        showPersons: false
    };
    // The event object is automatically passed by JavaScript.
    nameChangedHandler = (event, id) => {
        // We want to update the state of the person but only for one into which input
        // field we typed
        const personIndex = this.state.persons.findIndex(p => {
            return p.id === id;
        });

        // Not a good practice to mutate state directly. A person in persons array is a
        // JavaScript object. JavaScript objects are reference types. So we should not mutate
        // them directly bcz we only get a pointer when we reach put to person in the next
        // statement. Hence we would mutate the original object to which a pointer points.
        //const person = this.state.persons[personIndex];
        // Better approach (ES6)
        const person = {
            ...this.state.persons[personIndex]
        };
        // Alternative approach
        //const person = Object.assign({}, this.state.persons[personIndex]);
        // Manipulating the copied object, not the original one
        person.name = event.target.value;

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

        this.setState({ persons });
    };

    togglePersonsHandler = () => {
        const doesState = this.state.showPersons;
        // This will merge 'showPersons' attribute with state object
        this.setState({ showPersons: !doesState });
    };

    deletePersonHandler = personIndex => {
        //const persons = this.state.persons.slice(); //--> Approach 1
        const persons = [...this.state.persons]; // --> Approach 2 (ES6)
        persons.splice(personIndex, 1);
        this.setState({ persons });
    };

    render() {
        const style = {
            backgroundColor: "green",

```

```

color: "white",
font: "inherit",
border: "1px solid blue",
padding: "8px",
cursor: "pointer",
// using Radium feature
":hover": {
  backgroundColor: "lightgreen",
  color: "black"
}
};

let persons = null;
if (this.state.showPersons) {
  persons = (
    <div>
      {this.state.persons.map((person, index) => {
        return (
          <Person
            click={() => this.deletePersonHandler(index)}
            name={person.name}
            age={person.age}
            key={person.id}
            changed={event => this.nameChangedHandler(event, person.id)}
          />
        );
      })}
    </div>
  );
  style.backgroundColor = "red";
  // overriding hover
  style[":hover"] = {
    backgroundColor: "salmon",
    color: "black"
  };
}

// let classes = ['red', 'bold'].join() // "red bold"
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']
}

return (

```

```

<div className="App">
  <h1> Hello React!</h1>
  <p className={classes.join(" ")}> This is really working!</p>
  <button style={style} onClick={this.togglePersonsHandler}>
    Toggle persons
  </button>
  {persons}
</div>
);
}
export default Radium(App);

```

Radium does not limit you to use only pseudo selector, you can also use media query.

5.6. Using Radium for Media Queries

We can use MediaQuery like this in Person.css:

```

@media (min-width: 500px) {
  .Person {
    width: 450px;
  }
}

```

But we will see how to use it with Radium.

Person.js

```

import React from "react";
import "./Person.css";
import Radium from "radium";

const person = props => {
  // Using Radium
  const style = {
    "@media(min-width: 500px)": {
      width: "450px"
    }
  };

  // We need to import 'React' inorder to convert this code to JSX code
  (createElement())
  return (
    // style will override class bcz of CSS rules
    <div className="Person" style={style}>
      <p onClick={props.click}>

```

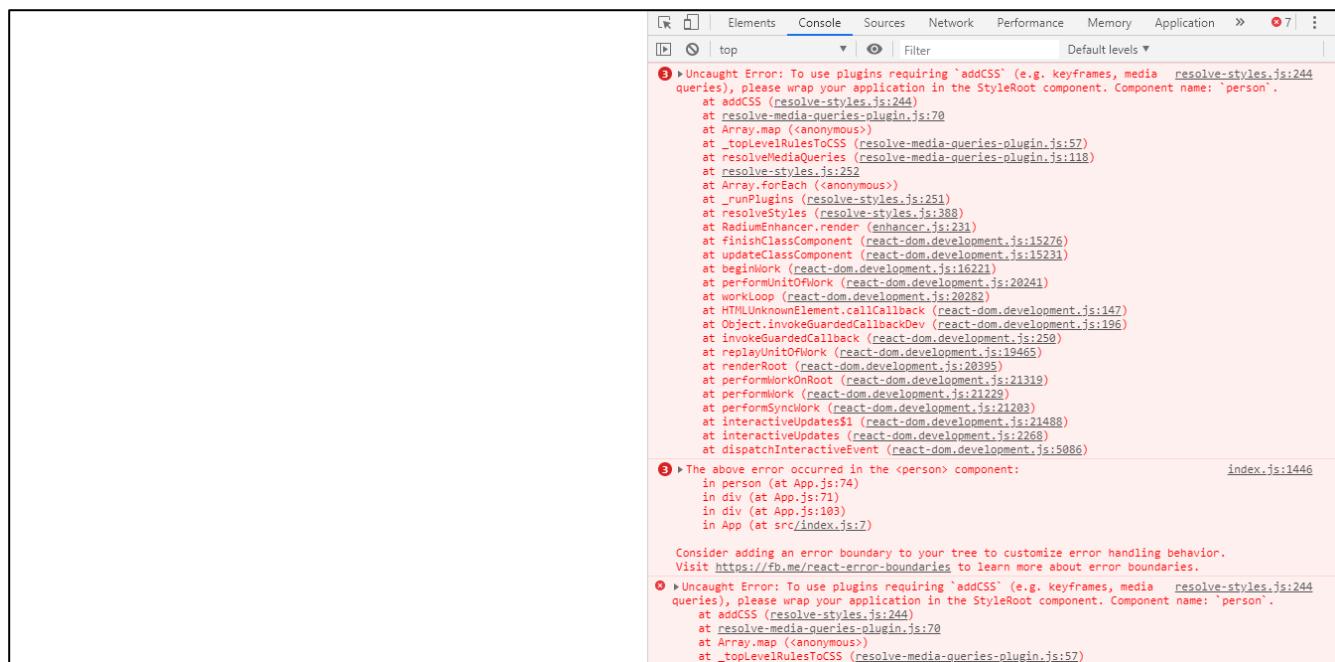
```

    I'm a person and My name is {props.name} and I am {props.age} years old
  </p>
  <p> {props.children}</p>
  /* onChange will change the value in paragraph.
'value' will take the changed value and display in the input box.
This is 2-way binding.
*/
  <input type="text" onChange={props.changed} value={props.name} />
</div>
);
};

export default Radium(person);

```

o/p:



The error asks to wrap component in `<StyleRoot>`. This is a component made available by radium. While wrapping the export with Radium is enough for pseudo-selectors, for media queries and keyframes we need to wrap our entire component with `<StyleRoot>`.

```
import Radium, { StyleRoot } from "radium";
```

Wrap everything in return statement with `<StyleRoot>`.

App.js

```

import React, { Component } from "react";
import "./App.css";
import Person from "./Person/Person.js";

```

```

import Radium, { StyleRoot } from "radium";

class App extends Component {
  state = {
    persons: [
      { id: "dudu", name: "Ram", age: "20" },
      { id: "wjow", name: "Shyam", age: "25" },
      { id: "ywey", name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
  };
  // The event object is automatically passed by javascript.
  nameChangedHandler = (event, id) => {
    // We want to update the state of the person but only for one into which input
    field we typed
    const personIndex = this.state.persons.findIndex(p => {
      return p.id === id;
    });

    // Not a good practice to mutate state directly. A person in persons array is a
    javascript object. Javascript objects are reference types. So we should not mutate
    them directly bcz we only get a pointer when we reach put to person in the next
    statement. Hence we would mutate the original object to which a pointer points.
    //const person = this.state.persons[personIndex];
    // Better approach (ES6)
    const person = {
      ...this.state.persons[personIndex]
    };
    // Alternative approach
    //const person = Object.assign({}, this.state.persons[personIndex]);
    // Manipulating the copied object, not the original one
    person.name = event.target.value;

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

    this.setState({ persons: persons });
  };

  togglePersonsHandler = () => {
    const doesState = this.state.showPersons;
    // This will merge 'showPersons' attribute with state object
    this.setState({ showPersons: !doesState });
  };

  deletePersonHandler = personIndex => {

```

```

//const persons = this.state.persons.slice(); //--> Approach 1
const persons = [...this.state.persons]; // --> Approach 2 (ES6)
persons.splice(personIndex, 1);
this.setState({ persons });
};

render() {
  const style = {
    backgroundColor: "green",
    color: "white",
    font: "inherit",
    border: "1px solid blue",
    padding: "8px",
    cursor: "pointer",
    // using Radium feature
    ":hover": {
      backgroundColor: "lightgreen",
      color: "black"
    }
  };
}

let persons = null;
if (this.state.showPersons) {
  persons = (
    <div>
      {this.state.persons.map((person, index) => {
        return (
          <Person
            click={() => this.deletePersonHandler(index)}
            name={person.name}
            age={person.age}
            key={person.id}
            changed={event => this.nameChangedHandler(event, person.id)}
          />
        );
      })}
    </div>
  );
  style.backgroundColor = "red";
  // overriding hover
  style[":hover"] = {
    backgroundColor: "salmon",
    color: "black"
  };
}

// let classes = ['red', 'bold'].join(); // "red bold"

```

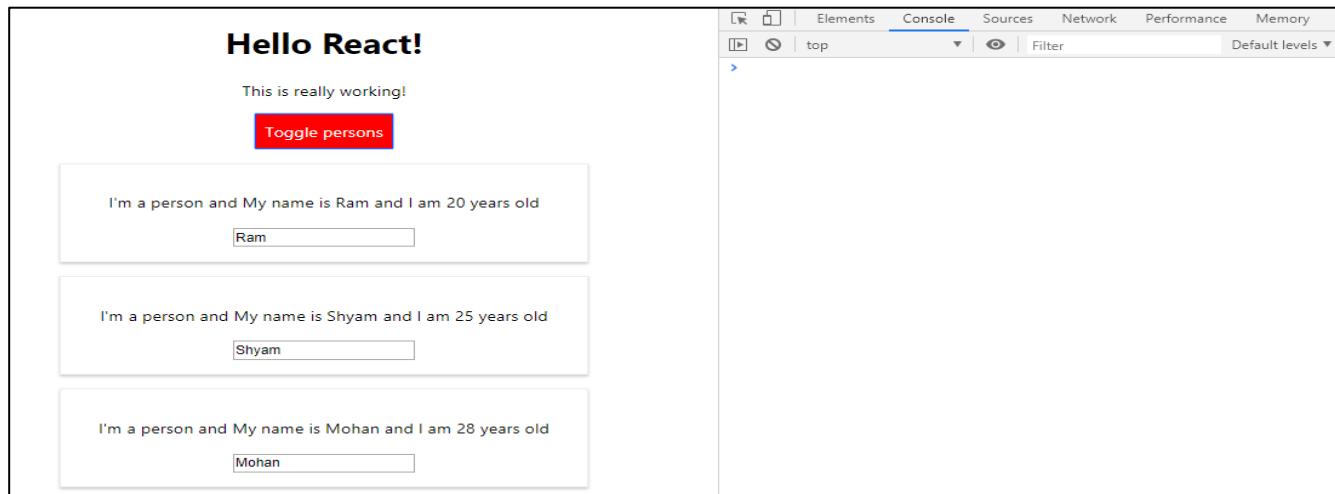
```

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']
}

return (
  <StyleRoot>
    <div className="App">
      <h1> Hello React!</h1>
      <p className={classes.join(" ")}> This is really working!</p>
      <button style={style} onClick={this.togglePersonsHandler}>
        Toggle persons
      </button>
      {persons}
    </div>
  </StyleRoot>
);
}
export default Radium(App);

```

O/P: (Error is gone).



5.7. Enabling & Using CSS Modules

It would be better if a css (Person.css) file could be scoped in js (Person.js) component file so that whatever style we have in css can be imported/assigned to elements in js component file and would not override styles in other components even if we were to share the names of the css classes.

We can use 'CSS Modules' for this.

- 1) Get rid of Radium import/export and style setup in person.js.
- 2) Do the same in App.js. Also remove the <StyleRoot>.

Now, we want to handle everything through scoped css file. To do this, we need to tweak the build configuration of our project.

In package.json, eject ('under scripts') gives access to build configurations. Use '[npm run eject](#)' to convert 'everything is managed for me' project to 'everything is managed for me but I can edit the configurations'. There is no way back after doing this.

```
E:\react-workspace\my-react-app> npm run eject
```

```
> my-react-app@0.1.0 eject E:\react-workspace\my-react-app
```

```
> react-scripts eject
```

NOTE: Create React App 2 supports TypeScript, Sass, CSS Modules and more without ejecting:
<https://reactjs.org/blog/2018/10/01/create-react-app-v2.html>

? Are you sure you want to eject? This action is permanent. Yes

Ejecting...

Copying files into E:\react-workspace\my-react-app

Adding \config\env.js to the project

Adding \config\paths.js to the project

Adding \config\webpack.config.js to the project

Adding \config\webpackDevServer.config.js to the project

Adding \config\jest\cssTransform.js to the project

Adding \config\jest\fileTransform.js to the project

Adding \scripts\build.js to the project

Adding \scripts\start.js to the project

Adding \scripts\test.js to the project

Updating the dependencies

Removing react-scripts from dependencies

- Adding @babel/core to dependencies
- Adding @svgr/webpack to dependencies
- Adding babel-core to dependencies
- Adding babel-eslint to dependencies
- Adding babel-jest to dependencies
- Adding babel-loader to dependencies
- Adding babel-plugin-named-asset-import to dependencies
- Adding babel-preset-react-app to dependencies
- Adding bfj to dependencies
- Adding case-sensitive-paths-webpack-plugin to dependencies
- Adding css-loader to dependencies
- Adding dotenv to dependencies
- Adding dotenv-expand to dependencies
- Adding eslint to dependencies
- Adding eslint-config-react-app to dependencies
- Adding eslint-loader to dependencies
- Adding eslint-plugin-flowtype to dependencies
- Adding eslint-plugin-import to dependencies
- Adding eslint-plugin-jsx-a11y to dependencies
- Adding eslint-plugin-react to dependencies
- Adding file-loader to dependencies
- Adding fs-extra to dependencies
- Adding html-webpack-plugin to dependencies
- Adding identity-obj-proxy to dependencies
- Adding jest to dependencies
- Adding jest-pnp-resolver to dependencies
- Adding jest-resolve to dependencies

Adding jest-watch-typeahead to dependencies
Adding mini-css-extract-plugin to dependencies
Adding optimize-css-assets-webpack-plugin to dependencies
Adding pnp-webpack-plugin to dependencies
Adding postcss-flexbugs-fixes to dependencies
Adding postcss-loader to dependencies
Adding postcss-preset-env to dependencies
Adding postcss-safe-parser to dependencies
Adding react-app-polyfill to dependencies
Adding react-dev-utils to dependencies
Adding resolve to dependencies
Adding sass-loader to dependencies
Adding style-loader to dependencies
Adding terser-webpack-plugin to dependencies
Adding url-loader to dependencies
Adding webpack to dependencies
Adding webpack-dev-server to dependencies
Adding webpack-manifest-plugin to dependencies
Adding workbox-webpack-plugin to dependencies
Updating the scripts
Replacing "react-scripts start" with "node scripts/start.js"
Replacing "react-scripts build" with "node scripts/build.js"
Replacing "react-scripts test" with "node scripts/test.js"
Configuring package.json
Adding Jest configuration
Adding Babel preset
Adding ESLint configuration

Running npm install...

audited 36158 packages in 40.491s

found 63 low severity vulnerabilities

run `npm audit fix` to fix them, or `npm audit` for details

Ejected successfully!

Please consider sharing why you ejected in this survey:

<http://goo.gl/forms/Bi6CZjk1EqsdelXk1>

After above command is completed, we can see two more folders (**config and scripts**) created.

(1) Scripts folder has one script for each command we have in package.json file.

(2) In config folder we have webpack.config.js file. Webpack is the bundling tool which applies all kinds of transformation/optimizations on our files and bundles them together. As part of this process, it also takes care of css files. [Webpack parses css import in JavaScript file](#). So Webpack is the place where we can adjust how we handle css files and where we can unlock this extra feature using 'css modules'.

(3) Changes in webpack.config.js.

```
loader: require.resolve("css-loader"),
  options: {
    cssOptions, // Kunj Changes for CSS modules
    modules: true,
    // To give unique hash to styles so that u do not override styles
    // across your application
    localIdentName: "[name]__[local]__[hash:base64:5]"
  }
```

Css loader is a tool which parses and converts our css, extracts it, stores it and imports css modules.
Restart npm server after the changes.

After the above change in Webpack.config.js, the css file (app.css) will be scoped in js (app.js) file. Below changes to be done in App.js;

1) `import classes from "./App.css";`

Any name can be given in place of 'classes'. 'classes' will be a JavaScript object containing css classes as properties. This change is done by cssloader. Cssloader converts css file into object.

2) `<div className="App">`

To be replaced with:

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

App.js

```
import React, { Component } from "react";
import classes from "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { id: "dudu", name: "Ram", age: "20" },
      { id: "wjom", name: "Shyam", age: "25" },
      { id: "ywey", name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
  };
  // The event object is automatically passed by javascript.
  nameChangedHandler = (event, id) => {
    // We want to update the state of the person but only for one into which input
    field we typed
    const personIndex = this.state.persons.findIndex(p => {
      return p.id === id;
    });

    // Not a good practice to mutate state directly. A person in persons array is a
    javascript object. Javascript objects are reference types. So we should not mutate
    them directly bcz we only get a pointer when we reach put to person in the next
    statement. Hence we would mutate the original object to which a pointer points.
    //const person = this.state.persons[personIndex];
    // Better approach (ES6)
    const person = {
      ...this.state.persons[personIndex]
    };
    // Alternative approach
    //const person = Object.assign({}, this.state.persons[personIndex]);
    // Manipulating the copied object, not the original one
    person.name = event.target.value;

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

    this.setState({ persons: persons });
  };

  togglePersonsHandler = () => {
```

```

const doesState = this.state.showPersons;
// This will merge 'showPersons' attribute with state object
this.setState({ showPersons: !doesState });
};

deletePersonHandler = personIndex => {
//const persons = this.state.persons.slice(); //--> Approach 1
const persons = [...this.state.persons]; // --> Approach 2 (ES6)
persons.splice(personIndex, 1);
this.setState({ persons: persons });
};

render() {
  const style = {
    backgroundColor: "green",
    color: "white",
    font: "inherit",
    border: "1px solid blue",
    padding: "8px",
    cursor: "pointer",
    // using Radium feature
    ":hover": {
      backgroundColor: "lightgreen",
      color: "black"
    }
  };
}

let persons = null;
if (this.state.showPersons) {
  persons = (
    <div>
      {this.state.persons.map((person, index) => {
        return (
          <Person
            click={() => this.deletePersonHandler(index)}
            name={person.name}
            age={person.age}
            key={person.id}
            changed={event => this.nameChangedHandler(event, person.id)}
          />
        );
      })}
    </div>
  );
  style.backgroundColor = "red";
  // overriding hover
  style[":hover"] = {

```

```

        backgroundColor: "salmon",
        color: "black"
    );
}

// let classes = ['red', 'bold'].join() // "red bold"
const assignedClasses = [];
if (this.state.persons.length <= 2) {
    assignedClasses.push(classes.red); // classes=['red']
}
if (this.state.persons.length <= 1) {
    assignedClasses.push(classes.bold); // classes=['red', 'bold']
}

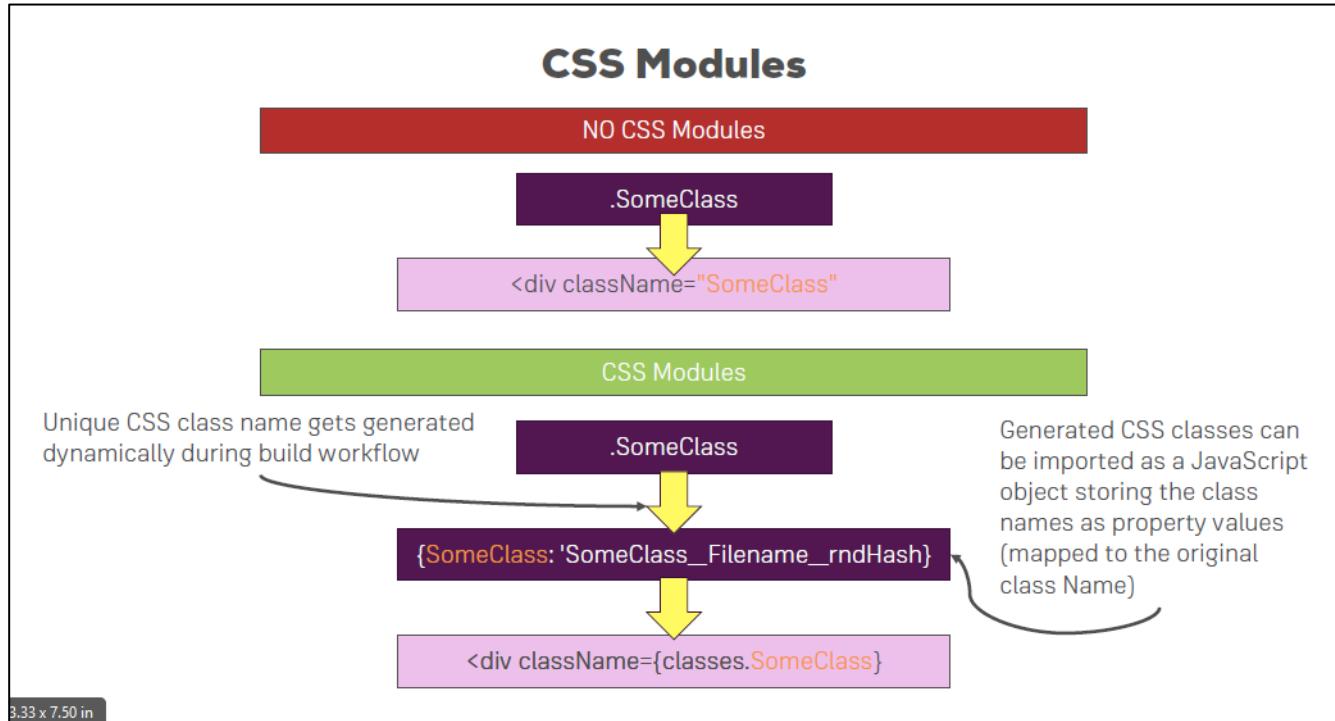
return (
    <div className={classes.App}>
        <h1> Hello React!</h1>
        <p className={assignedClasses.join(" ")}> This is really working!</p>
        <button style={style} onClick={this.togglePersonsHandler}>
            Toggle persons
        </button>
        {persons}
    </div>
);
}
export default App;

```

o/p:

The screenshot shows a browser window with the title 'Hello React!' and a message 'This is really working!'. Below it is a button labeled 'Toggle persons'. Underneath, there are three entries, each consisting of a name in a text input field and a corresponding message above it. The first entry is 'I'm a person and My name is Ram and I am 20 years old' with input 'Ram'. The second is 'I'm a person and My name is Shyam and I am 25 years old' with input 'Shyam'. The third is 'I'm a person and My name is Mohan and I am 28 years old' with input 'Mohan'. The browser's developer tools are open, specifically the 'Elements' tab, showing the DOM structure. A red box highlights the class 'App_App__3Ge-C' on the 'root' div. A callout annotation points to this class with the text: 'These names are created by CSSModule using the configuration set up done in webpack.server.js'. Another annotation at the bottom left of the dev tools panel says: 'This HTML file is a template. If you open it directly in the browser, you will see an empty page.' and 'You can add webfonts, meta tags, or analytics to this file. The build step will place the bundled scripts into the <body> tag.' At the bottom right, it says: 'To begin the development, run `npm start` or `yarn start`. To create a production bundle, use `npm run build` or `yarn build`.' followed by script tags for bundled files.

5.7.1 css-modules-learning-card.pdf



5.8. More on CSS Modules

CSS Modules are a relatively new concept (you can dive super-deep into them here: <https://github.com/css-modules/css-modules>). With CSS modules, you can write normal CSS code and make sure, that it only applies to a given component.

It's not using magic for that, instead it'll simply **automatically generate unique CSS class names** for you. And by importing a JS object and assigning classes from there, you use these dynamically generated, unique names. So the imported JS object simply exposes some properties which hold the generated CSS class names as values.

Example:

In Post.css File

```
.Post {  
  color: red;  
}
```

In Post Component File

```
import classes from './Post.css';  
  
const post = () => (
```

```
<div className={classes.Post}>...</div>
);
```

Here, `classes.Post` refers to an automatically generated `Post` property on the imported `classes` object. That property will in the end simply hold a value like `Post__Post__ah5_1`.

So your `.Post` class was automatically transformed to a different class (`Post__Post__ah5_1`) which is unique across the application. You also can't use it accidentally in other components because you don't know the generated string! You can only access it through the `classes` object. And if you import the CSS file (in the same way) in another component, the `classes` object there will hold a `Post` property which yields a **different** (!) CSS class name. Hence it's scoped to a given 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.

5.9. Adding Pseudo Selectors

Remove inline style from `<button>` and set the same style in ‘CSSModule’ way.

App.js

```
import React, { Component } from "react";
import classes from "./App.css";
import Person from "./Person/Person.js";

class App extends Component {
  state = {
    persons: [
      { id: "dudu", name: "Ram", age: "20" },
      { id: "wjom", name: "Shyam", age: "25" },
      { id: "ywey", name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
  };
  // The event object is automatically passed by javascript.
  nameChangedHandler = (event, id) => {
    // We want to update the state of the person but only for one into which input
    field we typed
```

```

const personIndex = this.state.persons.findIndex(p => {
  return p.id === id;
});

// Not a good practice to mutate state directly. A person in persons array is a
// javascript object. Javascript objects are reference types. So we should not mutate
// them directly bcz we only get a pointer when we reach put to person in the next
// statement. Hence we would mutate the original object to which a pointer points.
//const person = this.state.persons[personIndex];
// Better approach (ES6)
const person = {
  ...this.state.persons[personIndex]
};
// Alternative approach
//const person = Object.assign({}, this.state.persons[personIndex]);
// Manipulating the copied object, not the original one
person.name = event.target.value;

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

this.setState({ persons: persons });
};

togglePersonsHandler = () => {
  const doesState = this.state.showPersons;
  // This will merge 'showPersons' attribute with state object
  this.setState({ showPersons: !doesState });
};

deletePersonHandler = personIndex => {
  //const persons = this.state.persons.slice(); //--> Approach 1
  const persons = [...this.state.persons]; // --> Approach 2 (ES6)
  persons.splice(personIndex, 1);
  this.setState({ persons: persons });
};

render() {
  let persons = null;
  let buttonClass = "";
  if (this.state.showPersons) {
    persons = (
      <div>
        {this.state.persons.map((person, index) => {
          return (
            <Person
              click={() => this.deletePersonHandler(index)}

```

```

        name={person.name}
        age={person.age}
        key={person.id}
        changed={event => this.nameChangedHandler(event, person.id)}
      />
    );
  ){}
</div>
);
buttonClass = classes.Red;
}

// let classes = ['red', 'bold'].join() // "red bold"
const assignedClasses = [];
if (this.state.persons.length <= 2) {
  assignedClasses.push(classes.red); // classes=['red']
}
if (this.state.persons.length <= 1) {
  assignedClasses.push(classes.bold); // classes=['red', 'bold']
}

return (
  <div className={classes.App}>
    <h1> Hello React!</h1>
    <p className={assignedClasses.join(" ")}> This is really working!</p>
    <button className={buttonClass} onClick={this.togglePersonsHandler}>
      Toggle persons
    </button>
    {persons}
  </div>
);
}
}
export default App;

```

App.css

```

.App {
  text-align: center;
}

.red {
  color: red;
}

.bold {

```

```

    font-weight: bold;
}

/* for every button inside App class */

.App button {
  border: 1px solid blue;
  padding: 16px;
  background-color: green;
  font: inherit;
  color: white;
  cursor: pointer;
}

.App button:hover {
  background-color: lightgreen;
  color: black;
}

/* Style any button which has 'Red' class */

.App button.Red {
  background-color: red;
}

.App button.Red:hover {
  background-color: salmon;
  color: black;
}

```

5.10. Working with Media Queries

Person.js

```

import React from "react";
import classes from "./Person.css";

const person = props => {

  // We need to import 'React' inorder to convert this code to JSX code
  (createElement())
  return (
    <div className={classes.Person}>
      <p onClick={props.click}>
        I'm a person and My name is {props.name} and I am {props.age} years old
      </p>
    </div>
  );
}

```

```
<p> {props.children}</p>
/* onChange will change the value in paragraph.
'value' will take the changed value and display in the input box.
This is 2-way binding.
*/
<input type="text" onChange={props.changed} value={props.name} />
</div>
);
};
export default person;
```

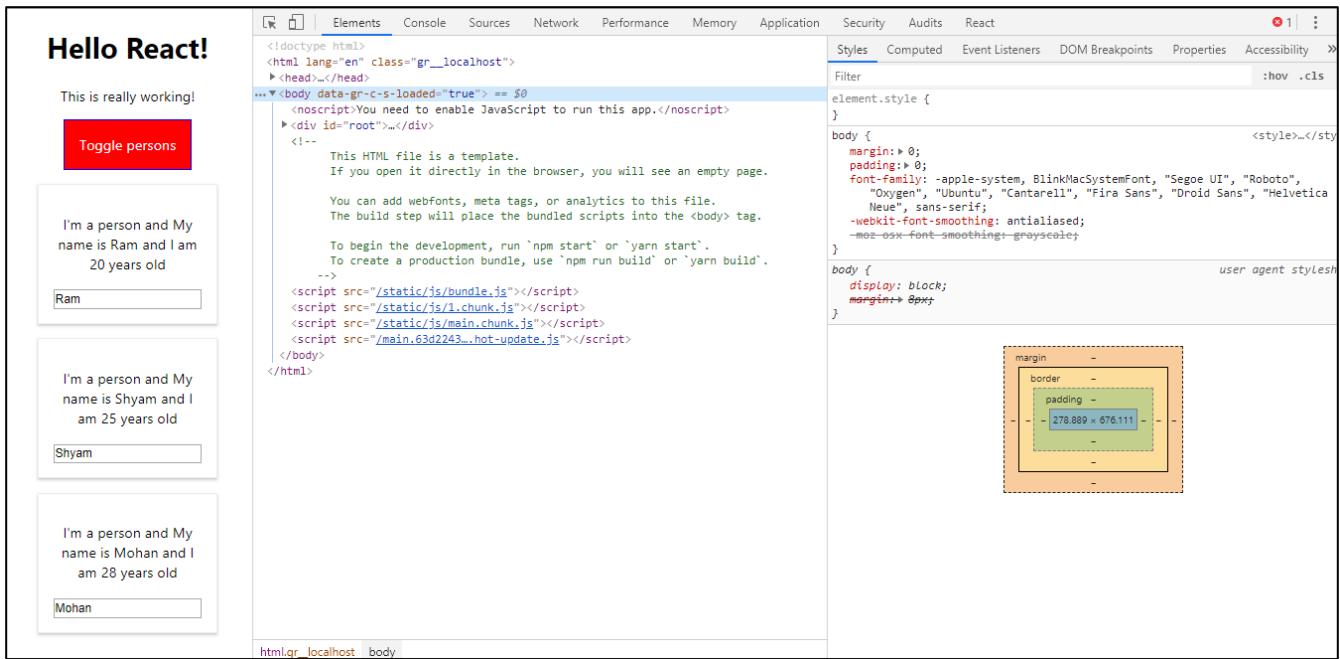
person.css

```
/* if we do not write .Person, this stylesheet will become global*/

.Person {
  width: 60%;
  margin: 16px auto;
  border: 1px solid #eee;
  box-shadow: 0 2px 3px #ccc;
  padding: 16px;
  text-align: center;
}

@media (min-width: 500px) {
  .Person {
    width: 450px;
  }
}
```

o/p after making the screen less wider than 500px:



5.11. Useful Resources & Links

- 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>

6. Debugging React Apps

6.1. Module Introduction

6.2. Understanding Error Messages

6.3. Finding Logical Errors by using Dev Tools & Sourcemap

6.4. Working with the React Developer Tools

Allows to understand the state of the components in app and manipulate it.

6.5. Using Error Boundaries (React 16+)

Sometimes you have code that might fail at runtime and you cannot guarantee that it always works. In this case you would probably want to show an error message to the users.

Example to throw an error randomly:

Person.js

```
import React from "react";
import classes from "./Person.css";

const person = props => {
  const rnd = Math.random();
  if (rnd > 0.7) {
    throw new Error("Something went wrong");
  }
  return (
    <div className={classes.Person}>
      <p onClick={props.click}>
        I'm a person and My name is {props.name} and I am {props.age} years old
      </p>
      <p> {props.children}</p>
      {/* onChange will change the value in paragraph.
      'value' will take the changed value and display in the input box.
      This is 2-way binding. */}
    );
};

export default person;
```

o/p after clicking 'toggle persons' will fluctuate between the correct one and the error page. The error page will look like this:

Error: Something went wrong

```
person
E:/react-workspace/my-react-app/src/Person/Person.js:7

  4 | const person = props => {
  5 |   const rnd = Math.random();
  6 |   if (rnd > 0.7) {
> 7 |     throw new Error("Something went wrong");
  8 |   }
  9 |   return (
10 |     <div className={classes.Person}>
```

[View compiled](#)

► 12 stack frames were collapsed.

This screen is visible only in development. It will not appear if the app crashes in production.
Open your browser's developer console to further inspect this error.

Q.) How to handle(catch) the error message and end the execution gracefully?

Ans: you can create ErrorBoundary components for this. The concept is ErrorBoundary however you can name your component as per your wish.

- 1) Create ErrorBoundary.js
- 2) Import ErrorBoundary.js into App.js
- 3) Wrap <Person> with <ErrorBoundary>. ErrorBoundary is a higher order component which simply wraps a component with the goal of handling any error that the component might throw.
- 4) Also provide key to <ErrorBoundary>

ErrorBoundary.js

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

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;
    }
}

export default ErrorBoundary;

```

App.js

```

import React, { Component } from "react";
import classes from "./App.css";
import Person from "./Person/Person.js";
import ErrorBoundary from "./ErrorBoundary/ErrorBoundary";

class App extends Component {
  state = {
    persons: [
      { id: "dudu", name: "Ram", age: "20" },
      { id: "wjom", name: "Shyam", age: "25" },
      { id: "ywney", name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
  };
  // The event object is automatically passed by javascript.
  nameChangedHandler = (event, id) => {
    // We want to update the state of the person but only for one into which input
    field we typed
    const personIndex = this.state.persons.findIndex(p => {
      return p.id === id;
    });

    // Not a good practice to mutate state directly. A person in persons array is a
    javascript object. Javascript objects are reference types. So we should not mutate
    them directly bcz we only get a pointer when we reach out to person in the next
    statement. Hence we would mutate the original object to which a pointer points.
    //const person = this.state.persons[personIndex];
    // Better approach (ES6)
    const person = {
      ...this.state.persons[personIndex]
    };
    // Alternative approach
    //const person = Object.assign({}, this.state.persons[personIndex]);
    // Manipulating the copied object, not the original one
    person.name = event.target.value;
  }
}

export default App;

```

```

const persons = [...this.state.persons];
persons[personIndex] = person;
this.setState({ persons: persons });
};

togglePersonsHandler = () => {
  const doesState = this.state.showPersons;
  // This will merge 'showPersons' attribute with state object
  this.setState({ showPersons: !doesState });
};

deletePersonHandler = personIndex => {
  //const persons = this.state.persons.slice(); //--> Approach 1
  const persons = [...this.state.persons]; // --> Approach 2 (ES6)
  persons.splice(personIndex, 1);
  this.setState({ persons: persons });
};

render() {
  let persons = null;
  let buttonClass = "";
  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>
    );
    buttonClass = classes.Red;
  }

  // let classes = ['red', 'bold'].join(); // "red bold"
  const assignedClasses = [];
  if (this.state.persons.length <= 2) {
    assignedClasses.push(classes.red); // classes=['red']
  }
  if (this.state.persons.length <= 1) {

```

```

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

    return (
        <div className={classes.App}>
            <h1> Hello React!</h1>
            <p className={assignedClasses.join(" ")}> This is really working!</p>
            <button className={buttonClass} onClick={this.togglePersonsHandler}>
                Toggle persons
            </button>
            {persons}
        </div>
    );
}

export default App;

```

o/p:

You still see the below error. This only happens during development mode. Once you build this for production and ship it to a real server, you will see what you render inside your ErrorBoundary. This does not mean that you should cluster your whole application with ErrorBoundary. Only use them when you have some code you know that it may fail. Do not wrap any other code bcz if normal code fails, you as a developer probably made a mistake during development. Only use ErrorBoundary for the cases you know that it might fail and you cannot control that. The react code will work fine and only the problematic component code will be replaced by your custom error message.

The screenshot shows a browser developer tools window with the 'Console' tab selected. An error message is displayed: "Error: Something went wrong". The error occurred at line 7 of Person.js, where a new Error object was thrown. The stack trace shows the error happened in the <h1> component of the App component. The browser also provides a link to learn more about error boundaries: <https://fb.me/react-error-boundaries>. There are also other error messages and stack traces visible in the console, related to invariant violations and invalid object types.

6.7. Useful Resources & Links

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

7. Diving Deeper into Components & React Internals

7.1. Module Introduction

Components are building blocks.

How we can split our component?

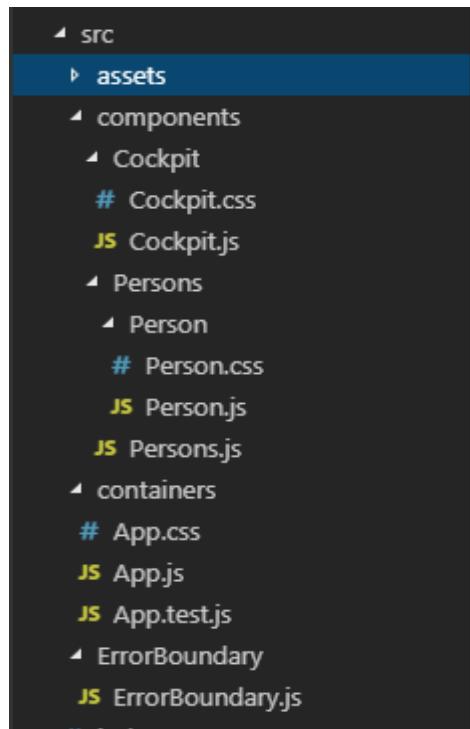
7.2. A Better Project Structure

You can split the application into more components. The question is what should go into its own component and what you group together in a higher component?

Person component looks focused. We do not need to split this up further into components.

In App.js, we are rendering persons. Components which also manage state should not be involved much with the UI rendering. In other words, their render() method should not contain too much JSX. It makes sense to create Person list component or a Persons component. We can then just pass array of persons into that component and inside that persons component we will do the mapping and render the list.

- 1) Create Persons folder and create a Persons.js file in it
- 2) Move Person folder into Persons folder
- 3) Create 'components' folder to store all components
- 4) Create 'assets' folder to store all images
- 5) Create 'containers' folder which holds all our containers, their css, testing files (App.js, App.css)
- 6) Create Cockpit folder to create Cockpit.js and to put UI code from App.js into it



7.3 Splitting an App Into Components

Persons.js can be a functional component as we do not have to maintain state in there.

Cockpit.js will hold all the UI code from App.js return statement.

App.js

```
import React, { Component } from "react";
import classes from "../containers/App.css";
import Persons from "../components/Persons/Persons";
import ErrorBoundary from "../ErrorBoundary/ErrorBoundary";
import Cockpit from "../components/Cockpit/Cockpit";

class App extends Component {
  state = {
    persons: [
      { id: "dudu", name: "Ram", age: "20" },
      { id: "wjom", name: "Shyam", age: "25" },
      { id: "ywey", name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
  };
  // The event object is automatically passed by javascript.
  nameChangedHandler = (event, id) => {
    // We want to update the state of the person but only for one into
    // which input field we typed
    const personIndex = this.state.persons.findIndex(p => {
      return p.id === id;
    });

    // Not a good practice to mutate state directly. A person in persons
    // array is a javascript object. Javascript objects are reference types. So we
    // should not mutate them directly bcz we only get a pointer when we reach put
    // to person in the next statement. Hence we would mutate the original object
    // to which a pointer points.
    //const person = this.state.persons[personIndex];
    // Better approach (ES6)
    const person = {
      ...this.state.persons[personIndex]
    };
    // Alternative approach
```

```

//const person = Object.assign({}, this.state.persons[personIndex]);
// Manipulating the copied object, not the original one
person.name = event.target.value;

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

this.setState({ persons: persons });
};

togglePersonsHandler = () => {
  const doesState = this.state.showPersons;
  // This will merge 'showPersons' attribute with state object
  this.setState({ showPersons: !doesState });
};

deletePersonHandler = personIndex => {
  //const persons = this.state.persons.slice(); //--> Approach 1
  const persons = [...this.state.persons]; // --> Approach 2 (ES6)
  persons.splice(personIndex, 1);
  this.setState({ persons: persons });
};

render() {
  let persons = null;
  let btnClass = "";
  if (this.state.showPersons) {
    persons = (
      <Persons
        persons={this.state.persons}
        clicked={this.deletePersonHandler}
        changed={this.nameChangedHandler}
      />
    );
  }

  // let classes = ['red', 'bold'].join(); // "red bold"
  const assignedClasses = [];
  if (this.state.persons.length <= 2) {
    assignedClasses.push(classes.red); // classes=['red']
  }
}

```

```

if (this.state.persons.length <= 1) {
  assignedClasses.push(classes.bold); // classes=['red', 'bold']
}

return (
  <div className={classes.App}>
    <Cockpit
      appTitle={this.props.title}
      showPersons={this.state.showPersons}
      persons={this.state.persons}
      clicked={this.togglePersonsHandler}
    />
    {persons}
  </div>
);
}
}

export default App;

```

Cockpit.js

```

import React from "react";
import classes from "./Cockpit.css";

const cockpit = props => {
  const assignedClasses = [];
  let btnClass = "";
  if (props.showPersons) {
    btnClass = classes.Red;
  }

  if (props.persons.length <= 2) {
    assignedClasses.push(classes.red); // classes = ['red']
  }
  if (props.persons.length <= 1) {
    assignedClasses.push(classes.bold); // classes = ['red', 'bold']
  }

  return (
    <div className={classes.Cockpit}>

```

```
<h1>{props.appTitle}</h1>
<p className={assignedClasses.join(" ")}>This is really working!</p>
<button className={btnClass} onClick={props.clicked}>
    Toggle Persons
</button>
</div>
);
};

export default cockpit;
```

Cockpit.css

```
.red {
    color: red;
}

.bold {
    font-weight: bold;
}

.Cockpit button {
    border: 1px solid blue;
    padding: 16px;
    background-color: green;
    font: inherit;
    color: white;
    cursor: pointer;
}

.Cockpit button:hover {
    background-color: lightgreen;
    color: black;
}

.Cockpit button.Red {
    background-color: red;
}

.Cockpit button.Red:hover {
    background-color: salmon;
```

```
    color: black;  
}
```

Person.js

```
import React from "react";  
import classes from "./Person.css";  
  
const person = props => {  
  return (  
    <div className={classes.Person}>  
      <p onClick={props.click}>  
        I'm a person and My name is {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;
```

Persons.js

```
import React from "react";  
import Person from "./Person/Person";  
  
const persons = props =>  
  props.persons.map((person, index) => {  
    return (  
      <Person  
        click={() => props.clicked(index)}  
        name={person.name}  
        age={person.age}  
        key={person.id}  
        changed={event => props.changed(event, person.id)}  
      />  
    );  
  });
});
```

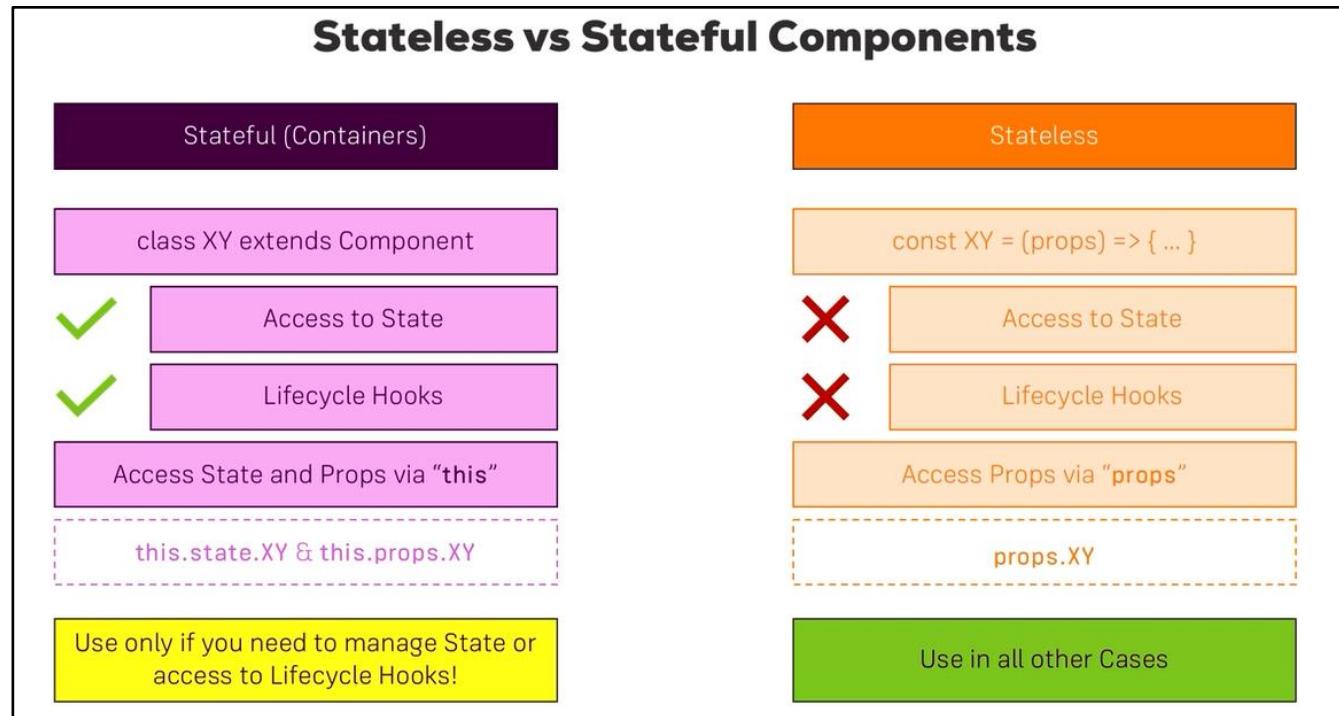
```
export default persons;
```

7.4. Comparing Stateless and Stateful Components

Q.) Try to create as many functional component as possible. Why?

Ans: these functions have a narrow focus and clear responsibility. They are only about presenting something. They cannot manage state.

As your application becomes bigger, it becomes harder to manage state.



We can even receive 'props' in Stateful components. It is automatically provided by component class and can be accessed by `this.props.propertyName`.

From React 16.8:

Stateful Components : Manage state (can be functional as well as class component)

Stateless Components : Does not Manage state (can be functional as well as class component)

Class-based vs Functional Components

class-based	Functional
class XY extends Component	const XY = props => { ... }
✓ Access to State	✓ Access to State (useState())
✓ Lifecycle Hooks	✗ Lifecycle Hooks
Access State and Props via "this"	Access Props via "props"
this.state.XY & this.props.XY	props.XY
Use if you need to manage State or access to Lifecycle Hooks and you don't want to use React Hooks!	Use in all other Cases

7.5. Understanding the Component Lifecycle

At the end when react creates a component for us (i.e. instantiates and renders for us), it runs through multiple lifecycle phases. We can define methods only in Stateful components which react will execute and will allow us to run some code during these lifecycle phases.

Component Lifecycle

Only available in Stateful Components!

constructor()

componentWillMount()

componentWillReceiveProps()

shouldComponentUpdate()

componentWillUpdate()

componentDidUpdate()

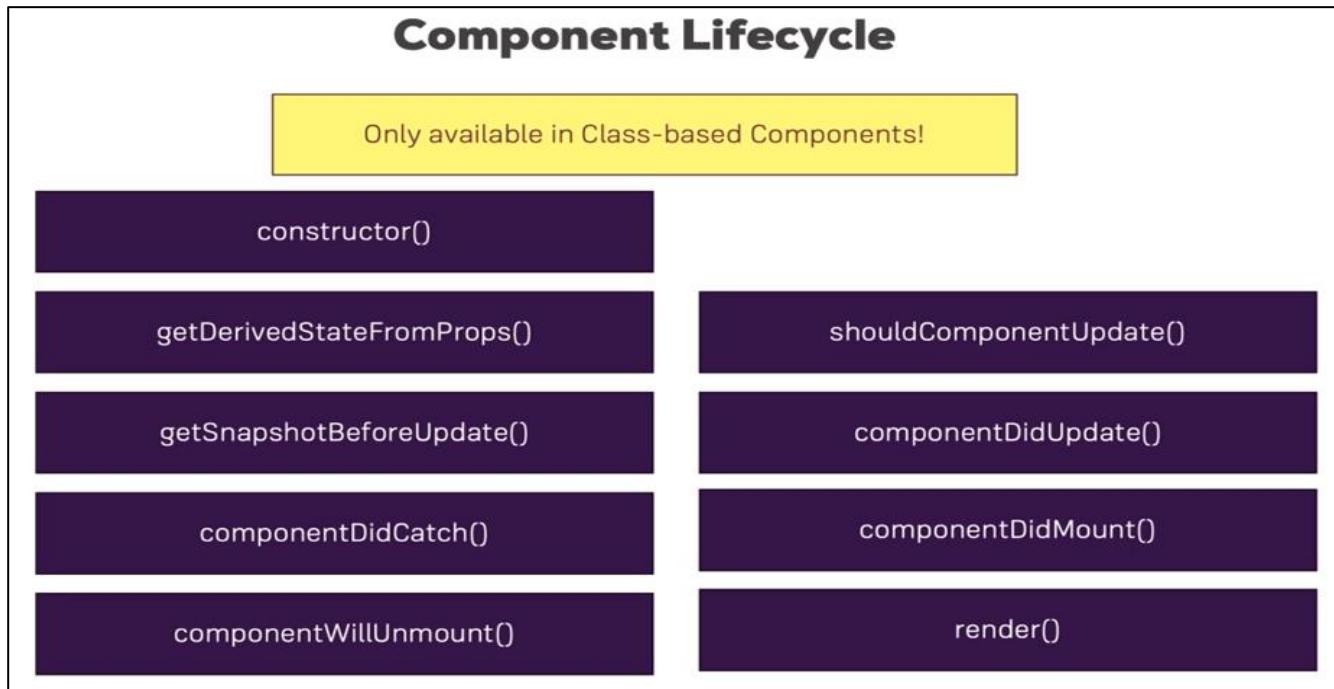
componentDidCatch()

componentDidMount()

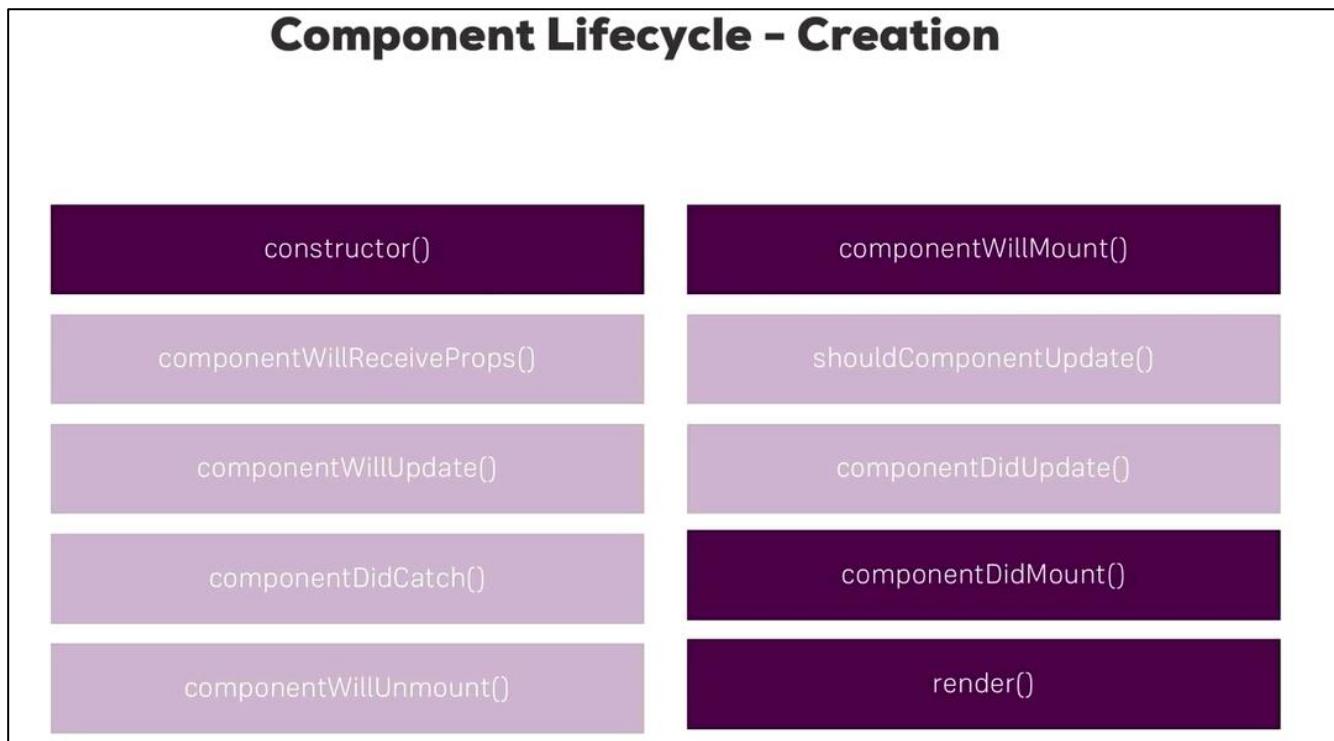
componentWillUnmount()

render()

From React 16.3 (Component Lifecycle):



Not all the above methods are executed during component creation. The methods executed are in the below screenshot:

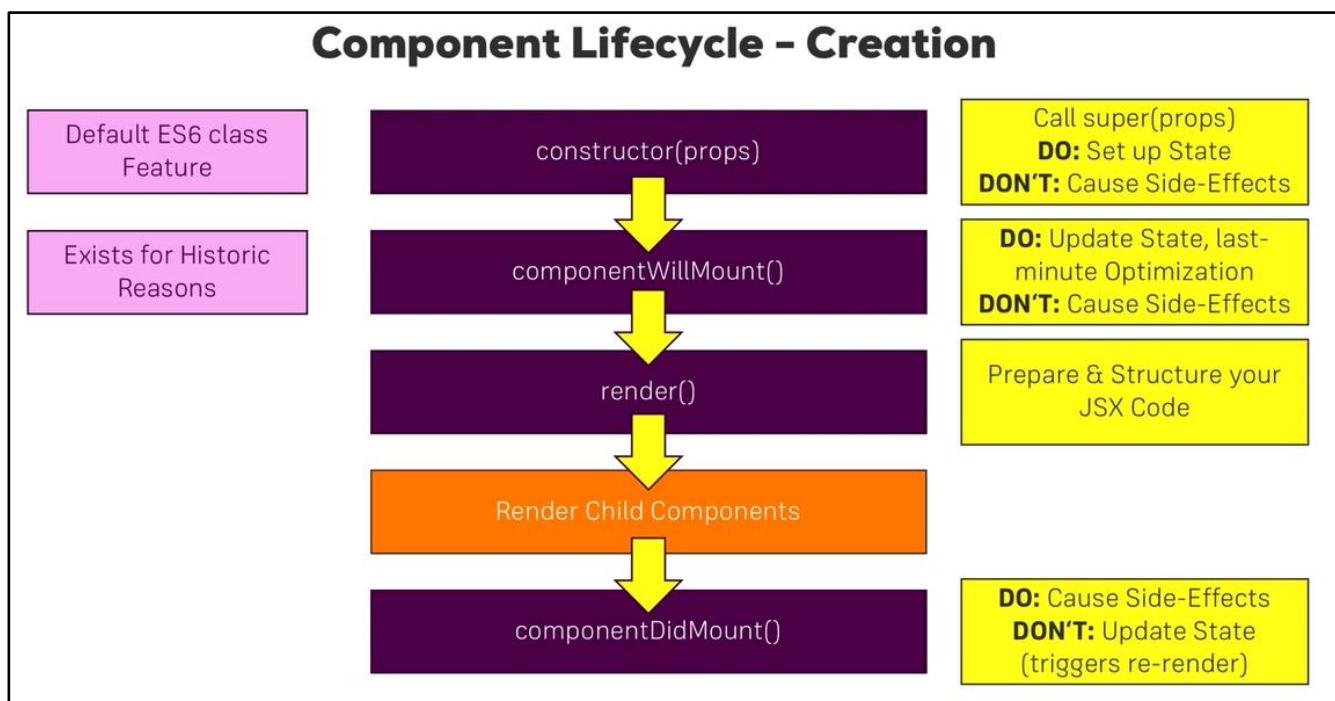


Component creation lifecycle process:

- First of all, **constructor** gets executed. Constructor is a default ES6 class feature. It is not a method defined by react. What react does is that it creates component at the end. It instantiates it and passes on any props this component receives to the constructor. If we do implement the `constructor()` method, we do have to call `super(props)`. This calls the constructor of the parent class. Since we can only implement this method in stateful component, the parent class is the component imported from react. By calling `super(props)`, this.props gets populated and managed by react. State can also be initialized in the constructor. ***Do not cause 'side effects' in constructor.***

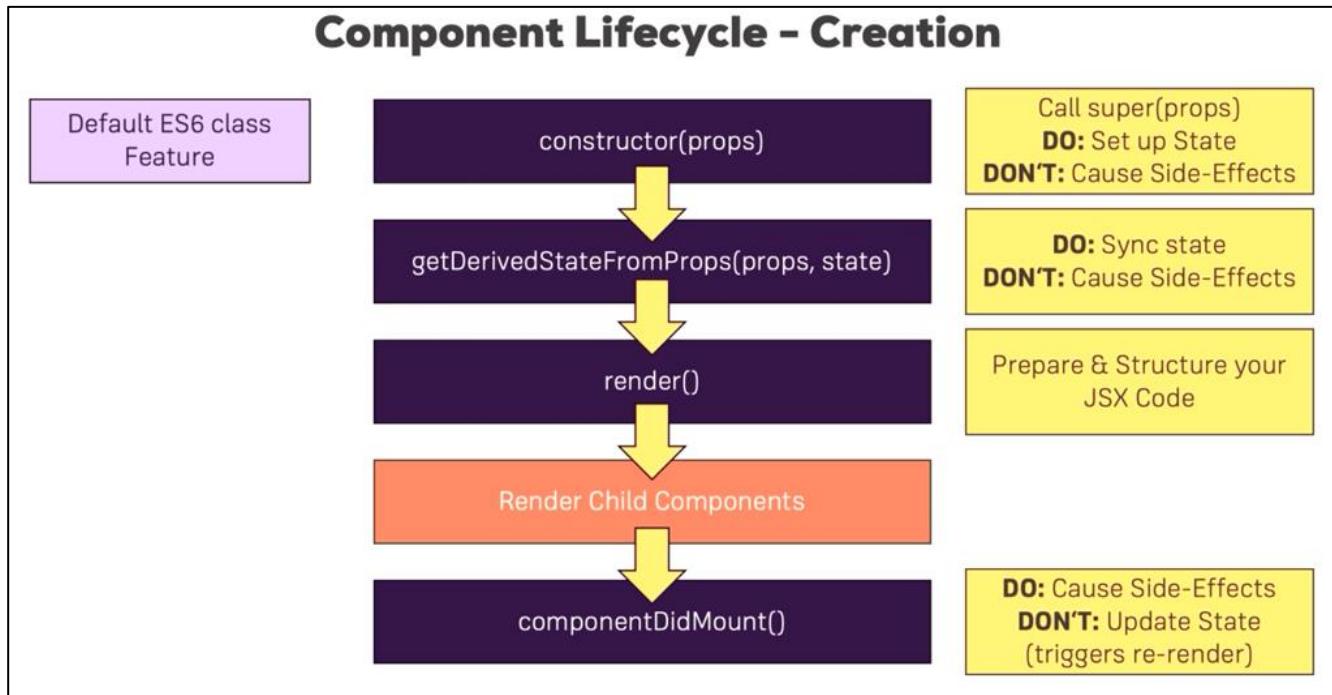
Side Effects → calling web url, updating state, storing something in local storage of the browser . This may lead to re-rendering of the application and hence cause performance issues. It may also lead to state becoming unpredictable.

- After `constructor()`, **componentWillMount()** gets executed. This method is defined by react. Scarcely used (Present only for historical reasons). **Used to update state for some last-minute optimizations.** ***Do not cause 'side effects' here.***
- render()** , Executing the `render()` does not mean that it accesses the real DOM. `Render()` gives idea to react how real DOM looks like. It defines how a component will look like from HTML perspective.
- render child components**
- componentDidMount()**. To tell that this component was successfully mounted. You can cause 'side-effects' here. ***Do not update state here as it will trigger re-render.*** Typically you will call http urls here.



From React 16.3:

getDerivedStateFromProps() → Whenever props change in class based components, you can sync state to them. Very rare use cases.



7.6. Converting Stateless to Stateful Components

Lifecycle methods can only be accessed in stateful components.

Persons.js

```
import React, { Component } from "react";
import Person from "./Person/Person";

class Persons extends Component {
  render() {
    return this.props.persons.map((person, index) => {
      return (
        <Person
          click={() => this.props.clicked(index)}
          name={person.name}
          age={person.age}
          key={person.id}
          changed={event => this.props.changed(event, person.id)}
        />
      );
    });
  }
}
```

```
}
```

```
export default Persons;
```

Person.js

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

import classes from "./Person.css";

class Person extends Component {
  render() {
    return (
      <div className={classes.Person}>
        <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}
        />
      </div>
    );
  }
}

export default Person;
```

7.7. Component Creation Lifecycle in Action

- 1) Create a constructor in App.js

Call super(props) as the first statement in the constructor to use this.props in the component. State can either be initialized outside constructor (state = {}) or inside constructor (this.state = {}). Outside constructor approach will only work on modern project setup.

- 2) Add lifecycle methods in App.js with console.log () to see the flow of execution.

App.js

```

import React, { Component } from "react";
import classes from "../containers/App.css";
import Persons from "../components/Persons/Persons";
import Cockpit from "../components/Cockpit/Cockpit";

class App extends Component {
  constructor(props) {
    super(props);
    console.log("[App.js] Inside Constructor", props);
    this.state = {
      persons: [
        { id: "dudu", name: "Ram", age: "20" },
        { id: "wjow", name: "Shyam", age: "25" },
        { id: "ywey", name: "Mohan", age: "28" }
      ],
      otherState: "some other value",
      showPersons: false
    };
  }

  componentWillMount() {
    console.log("[App.js] Inside componentWillMount()");
  }

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

  /*state = {
    persons: [
      { id: "dudu", name: "Ram", age: "20" },
      { id: "wjow", name: "Shyam", age: "25" },
      { id: "ywey", name: "Mohan", age: "28" }
    ],
    otherState: "Default state",
    showPersons: false
 };*/
  // The event object is automatically passed by javascript.
  nameChangedHandler = (event, id) => {
    // We want to update the state of the person but only for one into
    which input field we typed
  }
}

```

```

const personIndex = this.state.persons.findIndex(p => {
  return p.id === id;
});

// Not a good practice to mutate state directly. A person in persons
array is a javascript object. Javascript objects are reference types. So we
should not mutate them directly bcz we only get a pointer when we reach put
to person in the next statement. Hence we would mutate the original object
to which a pointer points.
//const person = this.state.persons[personIndex];
// Better approach (ES6)
const person = {
  ...this.state.persons[personIndex]
};
// Alternative approach
//const person = Object.assign({}, this.state.persons[personIndex]);
// Manipulating the copied object, not the original one
person.name = event.target.value;

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

this.setState({ persons: persons });
};

togglePersonsHandler = () => {
  const doesState = this.state.showPersons;
  // This will merge 'showPersons' attribute with state object
  this.setState({ showPersons: !doesState });
};

deletePersonHandler = personIndex => {
  //const persons = this.state.persons.slice(); //--> Approach 1
  const persons = [...this.state.persons]; // --> Approach 2 (ES6)
  persons.splice(personIndex, 1);
  this.setState({ persons: persons });
};

render() {
  console.log("[App.js] Inside render()");
  let persons = null;
}

```

```

if (this.state.showPersons) {
  persons = (
    <Persons
      persons={this.state.persons}
      clicked={this.deletePersonHandler}
      changed={this.nameChangedHandler}
    />
  );
}

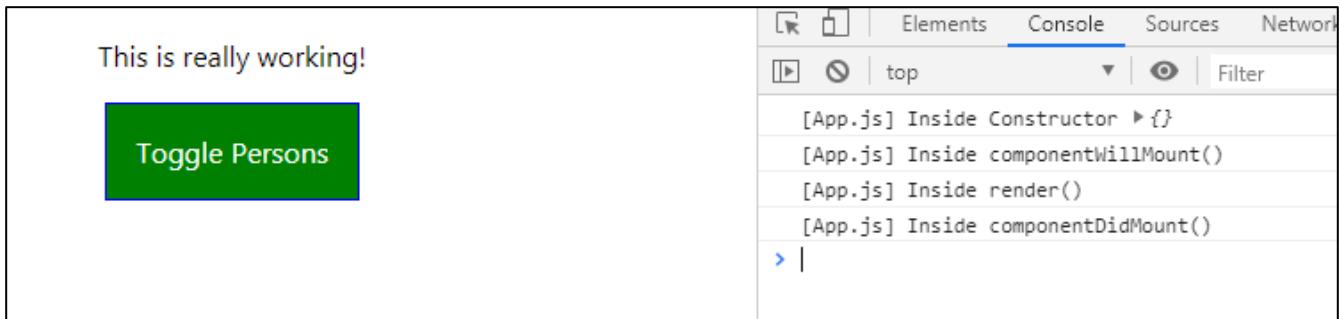
// let classes = ['red', 'bold'].join(); // "red bold"
const assignedClasses = [];
if (this.state.persons.length <= 2) {
  assignedClasses.push(classes.red); // classes=['red']
}
if (this.state.persons.length <= 1) {
  assignedClasses.push(classes.bold); // classes=['red', 'bold']
}

return (
  <div className={classes.App}>
    <Cockpit
      appTitle={this.props.title}
      showPersons={this.state.showPersons}
      persons={this.state.persons}
      clicked={this.togglePersonsHandler}
    />
    {persons}
  </div>
);
}
}

export default App;

```

o/p:



3) Add lifecycle methods in Persons.js with `console.log()` to see the flow of execution.

Persons.js

```

import React, { Component } from "react";
import Person from "./Person/Person";

class Persons extends Component {
  constructor(props) {
    super(props);
    console.log("[Persons.js] Inside Constructor", props);
  }

  componentWillMount() {
    console.log("[Persons.js] Inside componentWillMount()");
  }

  componentDidMount() {
    console.log("[Persons.js] Inside componentDidMount()");
  }

  render() {
    console.log("[Persons.js] Inside render()");
    return this.props.persons.map((person, index) => {
      return (
        <Person
          click={() => this.props.clicked(index)}
          name={person.name}
          age={person.age}
          key={person.id}
          changed={event => this.props.changed(event, person.id)}
        />
      );
    });
  }
}

```

```

    }
}

export default Persons;

```

O/P after clicking ‘toggle persons’: until you click ‘toggle persons’, you do not see lifecycle methods of Persons.js (You only see those of App.js) executed bcz we only show persons list conditionally.

- 4) Add lifecycle methods in Person.js with console.log () to see the flow of execution.

Person.js

```

import React, { Component } from "react";
import classes from "./Person.css";

class Person extends Component {
  constructor(props) {
    super(props);
    console.log("[Person.js] Inside Constructor", props);
  }

  componentWillMount() {
    console.log("[Person.js] Inside componentWillMount()");
  }

  componentDidMount() {
    console.log("[Person.js] Inside componentDidMount()");
  }
}

```

```

render() {
  console.log("[Person.js] Inside render()");
  return (
    <div className={classes.Person}>
      <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}
      />
    </div>
  );
}

export default Person;

```

O/P after clicking 'toggle persons': until you click 'toggle persons', you do not see lifecycle methods of Persons.js and Person.js (You only see those of App.js) executed bcz we only show persons list conditionally.

The screenshot shows a browser's developer tools with the 'Console' tab selected. On the left, there is a preview of the application interface displaying three person components with names Ram, Shyam, and Mohan, each with an input field. A red button labeled 'Toggle Persons' is visible. On the right, the 'Console' tab displays the following log entries:

```

[App.js] Inside Constructor > Object
[App.js] Inside componentWillMount()
[App.js] Inside render()
[App.js] Inside componentDidMount()
[App.js] Inside render()
[Persons.js] Inside Constructor > Object
[Persons.js] Inside componentWillMount()
[Persons.js] Inside render() ← This line is highlighted with a red box and has a blue arrow pointing to a note.
[Person.js] Inside Constructor > Object
[Person.js] Inside componentWillMount()
[Person.js] Inside render()
[Person.js] Inside Constructor > Object
[Person.js] Inside componentWillMount()
[Person.js] Inside render()
[Person.js] Inside Constructor > Object
[Person.js] Inside componentWillMount()
[Person.js] Inside render()
[Person.js] Inside componentDidMount() ← This line is highlighted with a green box and has a blue arrow pointing to a note.
[Persons.js] Inside componentDidMount() ← This line is highlighted with a green box and has a blue arrow pointing to a note.

```

Note: We do not see componentDidMount() after render() method, bcz child components will be rendered after render() of a component. As we have 3 child (Person.js), we see 3 person's lifecycle methods up to render().

Note: componentDidMount() 3 times for person.js

Note: componentDidMount() 1 time for Persons.js

7.8. componentWillUnmount()

We saw the Component Lifecycle during Component Creation in Action.

There also is one Lifecycle method which gets executed (**when implemented**) **right before** a Component is **removed** from the DOM: `componentWillUnmount()`.

Here's an Example:

App.js (using class App extends Component)

```
state = {  
    showUserComponent: true  
};  
  
removeUserHandler = () => {  
    this.setState({showUserComponent: false});  
}  
  
render() {  
    return (  
        <div>  
            {this.state.showUserComponent ? <User /> : null}  
            <button onClick={this.removeUserHandler}>Remove User Component</button>  
        </div>  
    );  
}
```

User.js (using class User extends Component)

```
componentWillUnmount() {  
    // Component is about to get removed => Perform any cleanup work here!  
    console.log('I\'m about to be removed!');  
}
```

In the above example, the User component is removed upon a button click (due to it being rendered conditionally and the condition result being changed to `false`). This triggers `componentWillUnmount()` to run in the User component right before the component is destroyed and removed from the DOM.

7.9. Component Updating Lifecycle Hooks (for props Changes)

Component lifecycle for updating. We need to differentiate between the update triggered by parent (by changing props) and internally triggered updates (by changing states).

Here we will look into update triggered by parent (props change).

(1) `componentWillReceiveProps(nextProps)`

We get the upcoming props as an argument here. We can **synchronize local state of our component to the props**. If you do not need to synchronize, you probably do not need to implement this method. **Do not cause side effects** as this will cause re-rendering of the component and hence cause side issue.

(2) `shouldComponentUpdate (nextProps, nextState)`

The two arguments are upcoming props and upcoming state. It receives the props and state that triggered this update.

This is a method which may actually cancel the updating process. If you return true here, the updating continues. In other methods you never return anything, but here you can return true or false.

If you return false, you save performance bcz react does not need to go through the whole component tree and call render and so on. Let's say you did allow component to update, you will reach componentWillUpdate (nextProps, nextState).

(3) componentWillUpdate (nextProps, nextState).

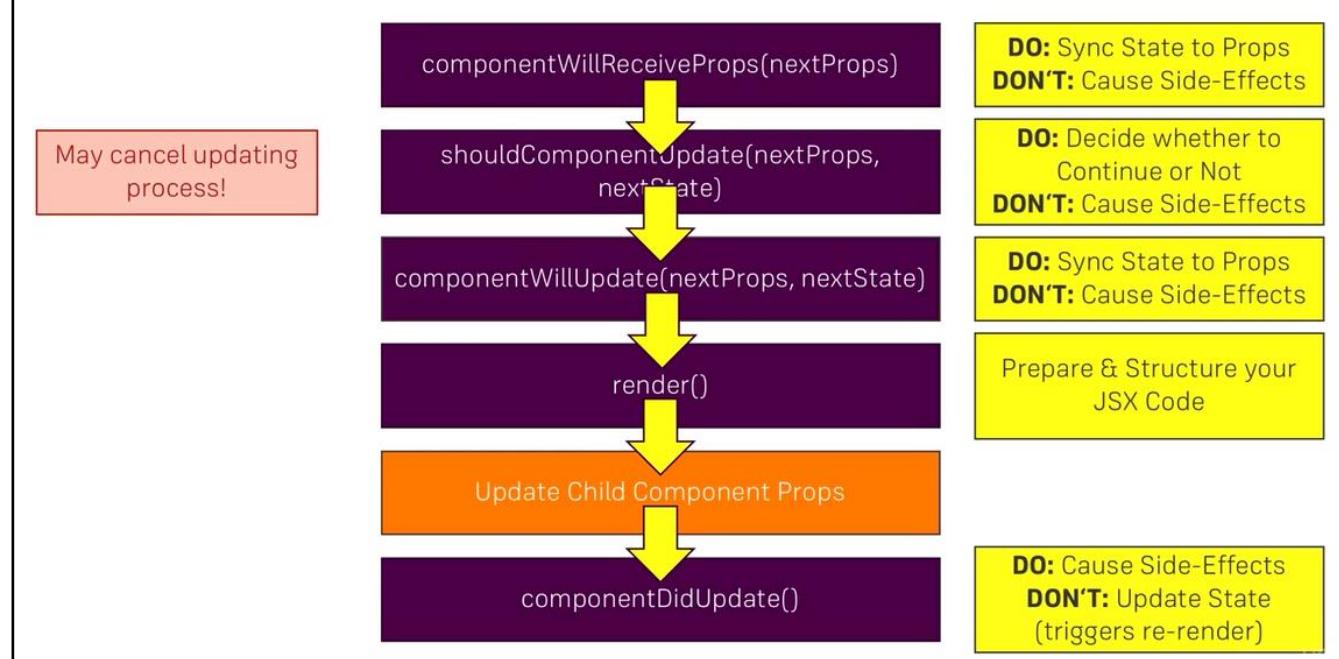
Better place to synchronize your state to props bcz unlike componentWillReceiveProps, here you know that you are going to continue updating.

(4) Render()

(5) Update child component props

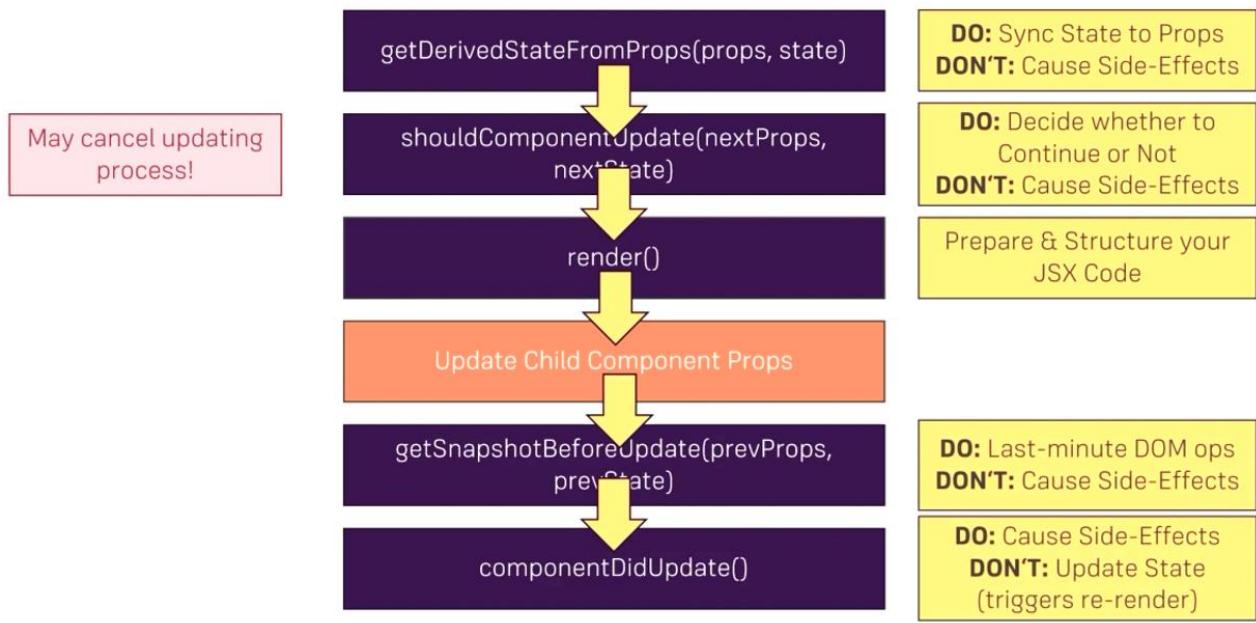
(6) componentDidUpdate() → Tells that the render() has been executed. You can **cause side effects**. You have to **watch out not to enter into infinite loop here**. If you make a http request and you get back a response and then update your component and then this cycle starts again. What you should not do here is outside of the then block of the promise of http request is updating the state with setState(). It is fine to do it as a result of some async task you are kicking off here, but you should not call it asynchronously bcz this will trigger unnecessary re-render.

Component Lifecycle – Update (triggered by Parent)



From React 16.3:

Component Lifecycle – Update



getDerivedStateFromProps(props, state) -> Used scarcely. To initialize the state of a component that updates based on props you are getting. Could be for ex for some form control which gets external properties and then you internally want to handle user input but initialize/update your state based on outside changes. Often there is more elegant way to update state based on props.

getDerivedStateFromProps is added as a safer alternative to the legacy **componentWillReceiveProps**

getSnapshotBeforeUpdate(prevProps, prevState) -> Scarcely used. Returns a snapshot object which you can freely configure. **Use it for last minute DOM operations (does not mean changes to the DOM)** like getting the current scrolling position of user. So imagine that your upcoming update of your component will re-render the DOM and will add new elements to the DOM and you therefore want to restore this scrolling position of the user once the update is done.

getSnapshotBeforeUpdate is added to support safely reading properties from e.g. the DOM before updates are made.

7.10. Component Updating Lifecycle in Action

Persons.js (Add **componentWillReceiveProps(nextProps)**)

```

componentWillReceiveProps(nextProps) {
  console.log(
    "[UPDATE Persons.js] Inside componentWillReceiveProps",
    nextProps
  );
}

```

O/P (see lifecycle methods):

The screenshot shows a browser's developer tools with the 'Console' tab selected. The output area displays the following log entries:

```
[App.js] Inside Constructor ►Object
[App.js] Inside componentWillMount()
[App.js] Inside render()
[App.js] Inside componentDidMount()
```

To the left of the console, there is a simple UI component consisting of a green button labeled "Toggle Persons" and a message "This is really working!".

O/P (after clicking 'Toggle Persons'): → Here also we do not see any update related lifecycle hook.

The screenshot shows a browser's developer tools with the 'Console' tab selected. The output area displays the following log entries, which are identical to the previous screenshot but include a red box highlighting the "Toggle Persons" button in the UI:

```
[App.js] Inside Constructor ►Object
[App.js] Inside componentWillMount()
[App.js] Inside render()
[App.js] Inside componentDidMount()
[App.js] Inside render()
[Persons.js] Inside Constructor ►{persons: Array(3), clicked: f, changed: f}
[Persons.js] Inside componentWillMount()
[Persons.js] Inside render()
[Person.js] Inside Constructor ►{click: f, name: "Ram", age: "20", changed: f}
[Person.js] Inside componentWillMount()
[Person.js] Inside render()
[Person.js] Inside render()
[Person.js] Inside Constructor ►{click: f, name: "Shyam", age: "25", changed: f}
[Person.js] Inside componentWillMount()
[Person.js] Inside render()
[Person.js] Inside render()
[Person.js] Inside Constructor ►{click: f, name: "Mohan", age: "28", changed: f}
[Person.js] Inside componentWillMount()
[Person.js] Inside render()
[Person.js] Inside render()
[Person.js] Inside render()
[Person.js] Inside componentDidMount()
[Persons.js] Inside componentDidMount()
```

To the left of the console, the UI now shows three separate boxes, each containing a message and an input field. The first box contains "I'm Ram and I am 20 years old!" and an input field with "Ram". The second box contains "I'm Shyam and I am 25 years old!" and an input field with "Shyam". The third box contains "I'm Mohan and I am 28 years old!" and an input field with "Mohan".

Clear console log and remove any person. Check O/P:

The screenshot shows a browser's developer tools with the 'Console' tab selected. The output area displays the following log entries, with a red arrow pointing to the "changed" event in the Persons.js code:

```
[App.js] Inside render()
[UPDATE Persons.js] Inside componentWillMountReceiveProps
  ▷{persons: Array(2), clicked: f, changed: f} ↴ Updated persons after deletion
    ▷changed: f (event, id)
    ▷clicked: f (personIndex)
    ▷persons: Array(2)
      ▷0: {id: "dudu", name: "Ram", age: "20"}
      ▷1: {id: "weye", name: "Mohan", age: "28"}
```

To the left of the console, the UI shows two boxes: "I'm Ram and I am 20 years old!" with input "Ram" and "I'm Mohan and I am 28 years old!" with input "Mohan".

Now add shouldComponentUpdate(nextProps, nextState):

Note: If you return false here, DOM will not update despite the fact that shouldComponentUpdate will be called and it will show the updated persons array.

Now add componentWillUpdate(nextProps, nextState)

Persons.js

```
import React, { Component } from "react";
import Person from "./Person/Person";

class Persons extends Component {
  constructor(props) {
    super(props);
    console.log("[Persons.js] Inside Constructor", props);
  }

  componentWillMount() {
    console.log("[Persons.js] Inside componentWillMount()");
  }

  componentDidMount() {
    console.log("[Persons.js] Inside componentDidMount()");
  }

  componentWillReceiveProps(nextProps) {
    console.log(
      "[UPDATE Persons.js] Inside componentWillReceiveProps",
      nextProps
    );
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log(
      "[UPDATE Persons.js] Inside shouldComponentUpdate",
      nextProps,
      nextState
    );
    return nextProps.persons !== this.props.persons;
    // return true;
  }

  componentWillUpdate(nextProps, nextState) {
    console.log(
      "[UPDATE Persons.js] Inside componentWillUpdate",
      nextProps,
      nextState
    );
  }
}
```

```

}

componentDidUpdate() {
  console.log("[UPDATE Persons.js] Inside componentDidUpdate");
}

render() {
  console.log("[Persons.js] Inside render()");
  return this.props.persons.map((person, index) => {
    return (
      <Person
        click={() => this.props.clicked(index)}
        name={person.name}
        age={person.age}
        key={person.id}
        changed={event => this.props.changed(event, person.id)}
      />
    );
  });
}

export default Persons;

```

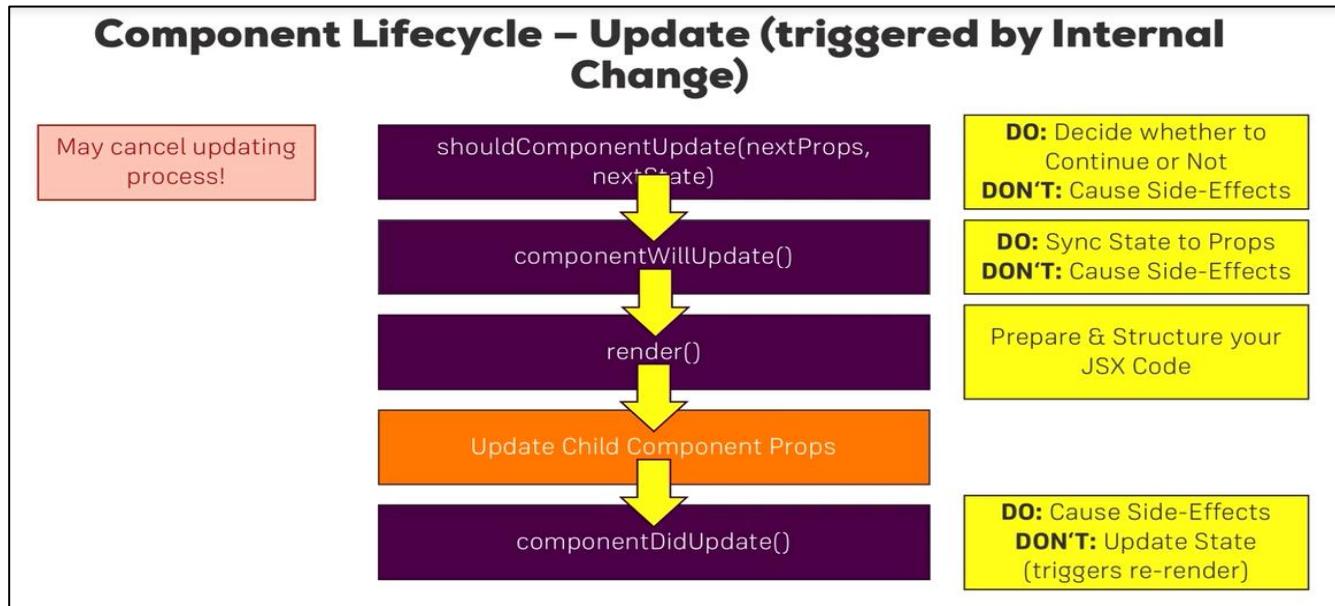
O/P after:

- (1) Click 'Toggle persons'
- (2) Delete console log
- (3) Remove one person

The screenshot shows a browser's developer tools with the 'Console' tab selected. The logs are as follows:

- [UPDATE Persons.js] Inside componentWillReceiveProps
- [App.js] Inside render()
 - persons: Array(2), clicked: f, changed: f
 - changed: f (event, id)
 - clicked: f (personIndex)
 - persons: Array(2)
 - 0: {id: "dudu", name: "Ram", age: "20"}
 - 1: {id: "wjoow", name: "Shyam", age: "25"}
 - length: 2
 - __proto__: Array(0)
 - __proto__: Object
- [UPDATE Persons.js] Inside shouldComponentUpdate
- [App.js] Inside render()
 - persons: Array(2), clicked: f, changed: f
 - changed: f (event, id)
 - clicked: f (personIndex)
 - persons: Array(2)
 - 0: {id: "dudu", name: "Ram", age: "20"}
 - 1: {id: "wjoow", name: "Shyam", age: "25"}
 - length: 2
 - __proto__: Array(0)
 - __proto__: Object
- [UPDATE Persons.js] Inside componentWillUpdate
- [Persons.js] Inside render()
- [Person.js] Inside render()
- [UPDATE Persons.js] Inside componentDidUpdate

7.11. Updating Lifecycle Hooks (Triggered by State Changes)



Implementing them in App.js:

Returning true for now from `shouldComponentUpdate()`:

```
shouldComponentUpdate(nextProps, nextState) {
  console.log(
    "[UPDATE App.js] Inside shouldComponentUpdate", nextProps, nextState
  );
  return true;
}
```

O/P (After deleting console and clicking ‘Toggle Persons’):

This screenshot shows a browser's developer tools console with the 'Console' tab selected. The output displays the state of the 'persons' array from the 'App.js' component. The array contains three objects, each representing a person with properties: id, name, age, and changed. The 'changed' property is explicitly set to true for the first two persons. The third person's 'changed' property is implicitly true because it is the result of a previous update. The 'otherState' variable is also shown as "some other value". The right side of the console lists the file names and line numbers for each log entry, corresponding to the code in 'App.js' and 'Person.js'. The 'Sources' tab is also visible at the top.

```
[UPDATE App.js] Inside shouldComponentUpdate > {}
  ↪ {persons: Array(3), otherState: "some other value", showPersons: true}
    ↪ otherState: "some other value"
  ↪ persons: Array(3)
    ↪ 0: {id: "dudu", name: "Ram", age: "20"}
    ↪ 1: {id: "wlow", name: "Shyam", age: "25"}
    ↪ 2: {id: "ywey", name: "Mohan", age: "28"}
      ↪ length: 3
    ↪ __proto__: Array(0)
  ↪ showPersons: true
  ↪ __proto__: Object

[UPDATE App.js] Inside componentWillUpdate > {}
  ↪ {persons: Array(3), otherState: "some other value", showPersons: true}

[App.js] Inside render()
[Persons.js] Inside Constructor > {persons: Array(3), clicked: f, changed: f}
[Persons.js] Inside componentWillMount()
[Persons.js] Inside render()
[Person.js] Inside Constructor > {click: f, name: "Ram", age: "20", changed: f}
[Person.js] Inside componentWillMount()
[Person.js] Inside render()
[Person.js] Inside Constructor > {click: f, name: "Shyam", age: "25", changed: f}
[Person.js] Inside componentWillMount()
[Person.js] Inside render()
[Person.js] Inside Constructor > {click: f, name: "Mohan", age: "28", changed: f}
[Person.js] Inside componentWillMount()
[Person.js] Inside render()
[Person.js] Inside render()

[Person.js] Inside componentDidMount()
[Persons.js] Inside componentDidMount()
[Person.js] Inside componentDidMount()
[Person.js] Inside componentDidMount()

[UPDATE App.js] Inside componentDidUpdate > {}

Default levels ▾
```

From React 16.3:

The screenshot shows a web browser window with the title "Person Manager". Inside the page, there is a message "This is really working!" and a red button labeled "Toggle Persons". Below this, there are three entries: "I'm Maxi and I am 28 years old!" with an input field containing "Maxi", "I'm Manu and I am 29 years old!" with an input field containing "Manu", and "I'm Stephanie and I am 26 years old!" with an input field containing "Stephanie". To the right of the page is the browser's developer tools console tab, which is active. The console shows the execution order of lifecycle methods for a component named "Person.js". The log entries are as follows:

- [App.js] getDerivedStateFromProps App.js:24
- ▶ {appTitle: "Person Manager"}
- [App.js] shouldComponentUpdate App.js:37
- [App.js] render App.js:77
- [Persons.js] shouldComponentUpdate Persons.js:16
- [Persons.js] rendering... Persons.js:35
- ③ [Person.js] rendering... Person.js:7
- [Persons.js] getSnapshotBeforeUpdate Persons.js:21
- [Persons.js] componentDidUpdate Persons.js:30
- ▶ {message: "Snapshot!"} Persons.js:31
- [App.js] componentDidUpdate App.js:42

Below the console, there are tabs for "Console", "What's New", "Coverage", "Search", and "Content scripts".

Lifecycle methods execution order for state changes:

getDerivedStateFromProps → shouldComponentUpdate → componentWillMount → render → componentDidUpdate.

With the introduction of hooks we can even manage state in functional components. There is 'useEffect' hook in functional component which is equivalent to lifecycle methods in class components. 'useEffect' is not a lifecycle hook, it is just a react hook (function) that you can add to functional components.

7.12. Performance Gains with PureComponents

Add a "Show persons" button above <cockpit> in App.js. This button just shows the persons.

```
<button onClick={() => { this.setState({ showPersons: true }) }}>Show  
Persons</button>
```

Return 'true' from 'shouldComponentUpdate()' in App.js and Persons.js.

Go to browser and click 'Show Persons' and it behaves as before. Clear the console but do not reload the page. Click 'Show Persons' again. We go through all the lifecycles again despite the fact that nothing changed. DO NOT mistake this for react re-rendering the actual DOM. It did not do that. You can confirm this by going to more tools → rendering → turn on 'Paint Flashing'. This is inefficient for

bigger apps. If you have many child nodes, going through all the render() methods even though nothing changed really can be a performance issue.

The screenshot shows a browser's developer tools with the 'Console' tab selected. The logs are as follows:

- [UPDATE App.js] Inside shouldComponentUpdate ()
- ↳ {persons: Array(3), otherState: "some other value", showPersons: true}
- ↳ otherState: "some other value"
- ↳ persons: Array(3)
 - ↳ 0: {id: "dudu", name: "Ram", age: "20"}
 - ↳ 1: {id: "wjom", name: "Shyam", age: "25"}
 - ↳ 2: {id: "ywey", name: "Mohan", age: "28"}
 - ↳ length: 3
 - ↳ __proto__: Array(0)
- ↳ showPersons: true
- ↳ __proto__: Object

Below this, there are several logs from 'Persons.js' and 'Person.js' related to componentWillUpdate, render, and componentDidUpdate.

DOM View:

A green button labeled 'Show Persons' is at the top. Below it, a red button labeled 'Toggle Persons'. Underneath are three white boxes containing text and input fields:

- I'm Ram and I am 20 years old!
Ram
- I'm Shyam and I am 25 years old!
Shyam
- I'm Mohan and I am 28 years old!
Mohan

That is why prior to that we have a check in shouldComponentUpdate() that tells if we really have the difference between the persons and only re-renders if it detects the difference. If any of the relevant property changes, update the DOM, else do not go there.

Persons.js

```
shouldComponentUpdate(nextProps, nextState) {
  console.log(
    "[UPDATE Persons.js] Inside shouldComponentUpdate",
    nextProps,
    nextState
  );
  return nextProps.persons !== this.props.persons ||
    nextProps.changed !== this.props.changed ||
    nextProps.clicked !== this.props.clicked;
  // return true;
}
```

O/P: ('show persons' → clear console → 'Show Persons')

The screenshot shows a browser's developer tools console tab. The logs are as follows:

- [UPDATE App.js] Inside shouldComponentUpdate ()
- ↳ {persons: Array(3), otherState: "some other value", showPersons: true}
- [UPDATE App.js] Inside componentWillUpdate ()
- ↳ {persons: Array(3), otherState: "some other value", showPersons: true}
- [App.js] Inside render()
- [UPDATE Persons.js] Inside componentWillReceiveProps ()
- ↳ {persons: Array(3), clicked: f, changed: f}
- [UPDATE Persons.js] Inside shouldComponentUpdate ()
- ↳ {persons: Array(3), clicked: f, changed: f} null
- [UPDATE App.js] Inside componentDidUpdate ()

A red arrow points to the final log entry in Persons.js: "No render() for individual Person component".

No render() method called above as we detected no change in Person component. Add a check in App.js.

App.js

```
shouldComponentUpdate(nextProps, nextState) {
  console.log(
    "[UPDATE App.js] Inside shouldComponentUpdate",
    nextProps,
    nextState
  );
  return nextState.persons !== this.state.persons ||
    nextState.showPersons !== this.state.showPersons;
}
```

O/P: ('show persons' → clear console → Show persons)

We only reach shouldComponentUpdate() in App.js.

The screenshot shows a React application with three components. Each component displays a message like "I'm Ram and I am 20 years old!" and contains an input field with the name. A "Toggle Persons" button is at the top. The browser's developer tools show the state of the App.js component, which includes an array of persons (dudu, shyam, mohan) and a showPersons boolean.

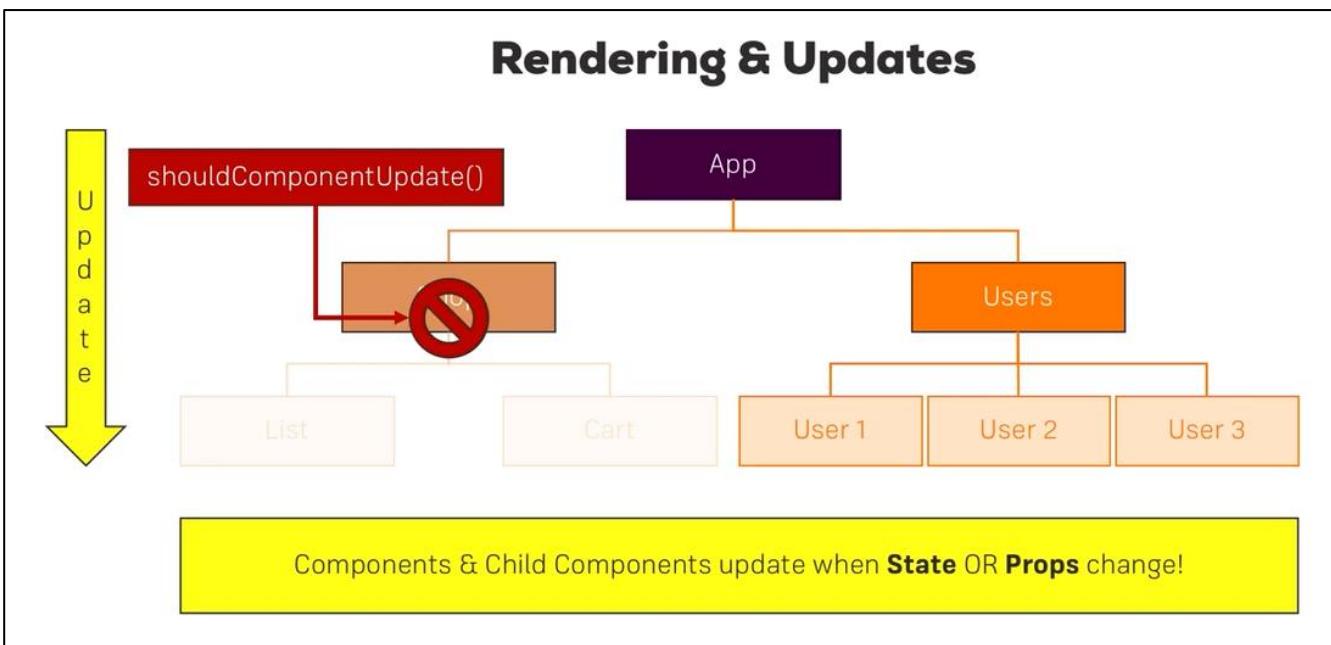
If you want to make a [shallow check](#) as we are doing in `shouldComponentUpdate()`, we do not need to implement `shouldComponentUpdate()`. We can comment this and inherit from a different type of component '**PureComponent**'. Make changes to `person.js` and `App.js`. if you want to check all the props for change, use `PureComponent` instead of `shouldcomponentUpdate()`.

Q) Should you always use PureComponent for your entire application?

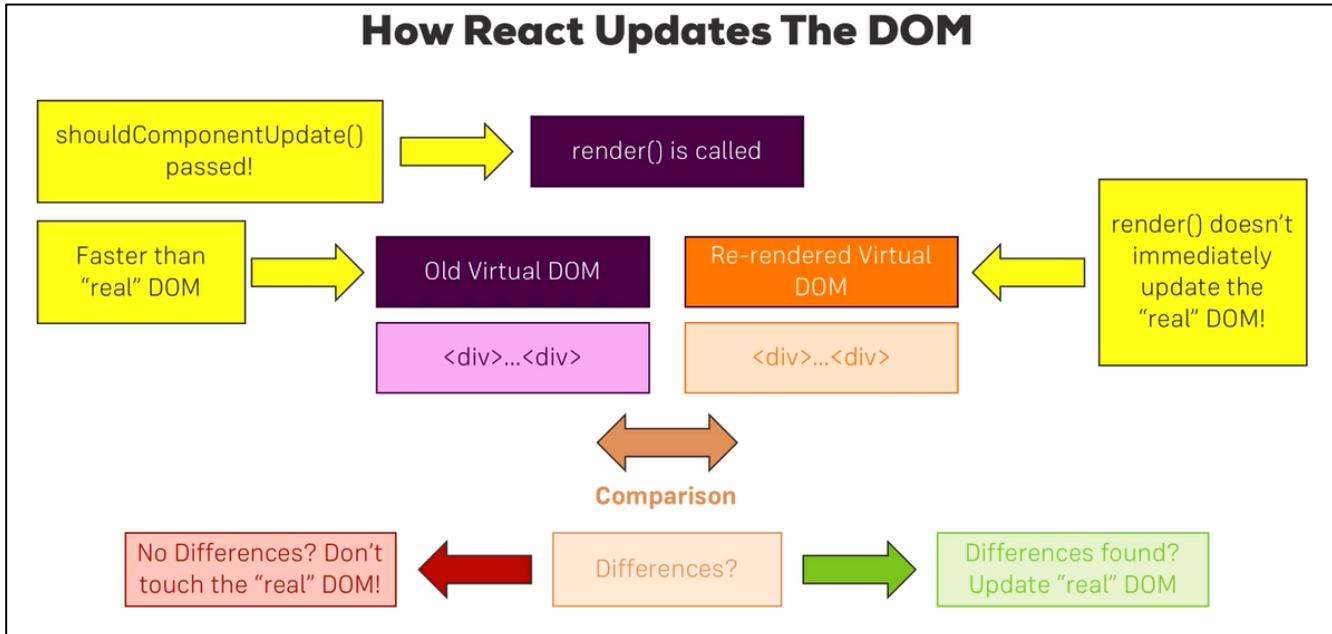
Ans: No. You should only use it when you know that the updates might not be required.

`PureComponent` implements `shouldComponentUpdate()` for you. So it takes a bit of performance.

7.13. How React Updates the App & Component Tree



7.14. Understanding React's DOM Updating Strategy



The `render()` method does not immediately render it to real DOM. Render is more of a suggestion, how HTML should look like at the end, but render can very well be called and lead to the same result as is already displayed. This is part of the reason why we use `shouldComponentUpdate()` to prevent unnecessary render call. It compares old virtual DOM and re-rendered (new) virtual DOM and checks if there are differences. If it detects the differences, it reaches out to the real DOM and updates it and even then it does not re-render the real DOM entirely. It only changes the places where differences were detected. For ex, if a button text changes, it will only update the text, not re-render the entire button. If no differences were found, it does not touch the real DOM, render did execute and comparison was made so `shouldComponentUpdate()` might make sense to prevent this if this is not needed.

Re-rendered DOM is created when `render()` method is called.

7.15. Returning Adjacent Elements (React 16+)

Our components typically have a wrapping element. You must not have multiple elements sitting next to each other on root level inside return statement of your `render()` method. Example:

Person.js

```
render() {
  console.log("[Person.js] Inside render()");
  return (
    <div className={classes.Person}>
      <p onClick={this.props.click}>
        I'm {this.props.name} and I am {this.props.age} years old!
      </p>
    </div>
  );
}
```

```

<p>{this.props.children}</p>
<input
  type="text"
  onChange={this.props.changed}
  value={this.props.name}
/>
</div>
);
}

```

But if it is an array of elements, it works.

Persons.js

```

render() {
  console.log("[Persons.js] Inside render()");
  return this.props.persons.map((person, index) => {
    return (
      <Person
        click={() => this.props.clicked(index)}
        name={person.name}
        age={person.age}
        key={person.id}
        changed={event => this.props.changed(event, person.id)}
      />
    );
  );
}

```

Even in Person.js, you can return an array of 3 html elements. Each array item needs to have a key then. However you can achieve this, you will lose the style that we used in wrapping <div> approach. Most of the times you will have code wrapped in <div>, the exception being when you need to return list.

Person.js

```

render() {
  console.log("[Person.js] Inside render()");
  // return (
  //   <div className={classes.Person}>
  //     <p onClick={this.props.click}>
  //       I'm {this.props.name} and I am {this.props.age} years old!
  //     </p>
  //   </div>
}

```

```

//      </p>
//      <p>{this.props.children}</p>
//      <input
//          type="text"
//          onChange={this.props.changed}
//          value={this.props.name}
//      />
//    </div>
//  );
}

return [
  <p key="1" onClick={this.props.click}>I'm {this.props.name} and I am
  {this.props.age} years old!</p>,
  <p key="2">{this.props.children}</p>,
  <input key="3" type="text" onChange={this.props.changed}
  value={this.props.name} />
]
}

```

In Cockpit.js we have,

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

We can define our own button class and get rid of above style.

Cockpit.css

```

.red {
  color: red;
}

.bold {
  font-weight: bold;
}

.Button {
  border: 1px solid blue;
  padding: 16px;
  background-color: green;
  font: inherit;
  color: white;
}

```

```

        cursor: pointer;
    }

.Button:hover {
    background-color: lightgreen;
    color: black;
}

.Button.Red {
    background-color: red;
}

.Button.Red:hover {
    background-color: salmon;
    color: black;
}

```

Cockpit.js

```

import React from "react";
import classes from "./Cockpit.css";

const cockpit = props => {
    const assignedClasses = [];
    let btnClass = classes.Button;
    if (props.showPersons) {
        btnClass = [classes.Button, classes.Red].join(' ');
    }

    if (props.persons.length <= 2) {
        assignedClasses.push(classes.red); // classes = ['red']
    }
    if (props.persons.length <= 1) {
        assignedClasses.push(classes.bold); // classes = ['red', 'bold']
    }

    return (
        <div>
            <h1>{props.appTitle}</h1>
            <p className={assignedClasses.join(" ")}>This is really working!</p>
            <button className={btnClass} onClick={props.clicked}>

```

```

        Toggle Persons
    </button>
</div>
);
};

export default cockpit;

```

As there is no class in <div> of return(), we can return html elements as lists. Another alternative is higher order component.

Create a folder 'hoc' to store higher order component. **Higher order components** are normal react components with one speciality that they are not representational but they wrap other component to add certain functionality.

Create Aux.js. This component take props and return `some JSX (props.children)

Import Aux.js in cockpit.js.

Wrap content inside render() with <Aux>.

Aux.js

```

const aux = (props) => props.children;

export default aux;

```

cockpit.js

```

import React from "react";
import classes from "./Cockpit.css";
import Aux from "../..../hoc/Aux";

const cockpit = props => {
    const assignedClasses = [];
    let btnClass = classes.Button;
    if (props.showPersons) {
        btnClass = [classes.Button, classes.Red].join(' ');
    }

    if (props.persons.length <= 2) {
        assignedClasses.push(classes.red); // classes = ['red']
    }
    if (props.persons.length <= 1) {

```

```

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

return (
<Aux>
  <h1>{props.appTitle}</h1>
  <p className={assignedClasses.join(" ")}>This is really working!</p>
  <button className={btnClass} onClick={props.clicked}>
    Toggle Persons
  </button>
</Aux>
);
};

export default cockpit;

```

7.16. React 16.2 Feature Fragments

If your project uses **React 16.2**, you can now use a built-in "Aux" component - a so called **fragment**.

It's actually not called Aux but you simply use <> - an empty JSX tag.

So the following code

```
<Aux>
  <h1>First Element</h1>
  <h1>Second Element</h1>
</Aux>
```

becomes

```
<>
  <h1>First Element</h1>
  <h1>Second Element</h1>
</>
```

Behind the scenes, it does the same our `Aux` component did.

7.17. Understanding Higher Order Components (HOCs)

Create `WithClass.js` in 'hoc' folder.

`WithClass.js`

```
import React from 'react';
```

```

const withClass = (props) => (
  <div className= {props.classes}>
    {props.children}
  </div>
);

export default withClass;

```

Person.js

```

render() {
  console.log("[Person.js] Inside render()");
  return (
    <WithClass classes={classes.Person}>
      <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}
      />
    </WithClass>
  );
}

```

7.18. Windows Users Must Read - File Downloads

On Windows, the Aux.js filename is not allowed in ZIP archives. Hence when extracting the attached source code, you might get prompted to rename the Aux.js file. You might also **face difficulties creating an Aux folder** and Aux.js file.

I really apologize for that inconvenience, Windows is really doing an amazing job here ;-).

Follow these fixes:

1) Problems when unzipping the attached file:

Simply **skip this step** (e.g. by pressing "No") and **ignore** the upcoming error message.

In the extracted folder, you'll then find **all source files** EXCEPT for the Aux.js file. In later course modules (where we work on the course project), the Aux.js file can be found in an Aux/ subfolder inside hoc/.

Make sure to take the Aux.js file **attached to this lecture** and place it inside the hoc/ or hoc/Aux/ folder (which ever of the two you got).

2) Problems with the creation of an Aux folder and/ or file:

Simply name both differently. For example, you may create an Auxiliary folder and name the file inside of it Auxiliary.js. Make sure to then adjust your imports (import Aux from './path/to/Auxiliary/Auxiliary') and you should be fine.

7.18.1 Aux.js

7.18.2 Auxiliary.js

7.19. A Different Approach to HOCs

Writing withClass higher order component differently.

withClass.js

```
import React from 'react';
//Takes a configuration and returns a function
const withClass = (WrappedComponent, className) => {
  return (props) => (<div className={className}>
// never manipulate your WrappedComponent here, just use it
    <WrappedComponent />
  </div>
)
}

export default withClass;
```

App.js

```
return (
  <Aux>
    <button onClick={() => { this.setState({ showPersons: true }) }>Show Persons</button>
    <Cockpit
      appTitle={this.props.title}
      showPersons={this.state.showPersons}
```

```
        persons={this.state.persons}
        clicked={this.togglePersonsHandler}
    />
    {persons}
  </Aux>
);
}
}

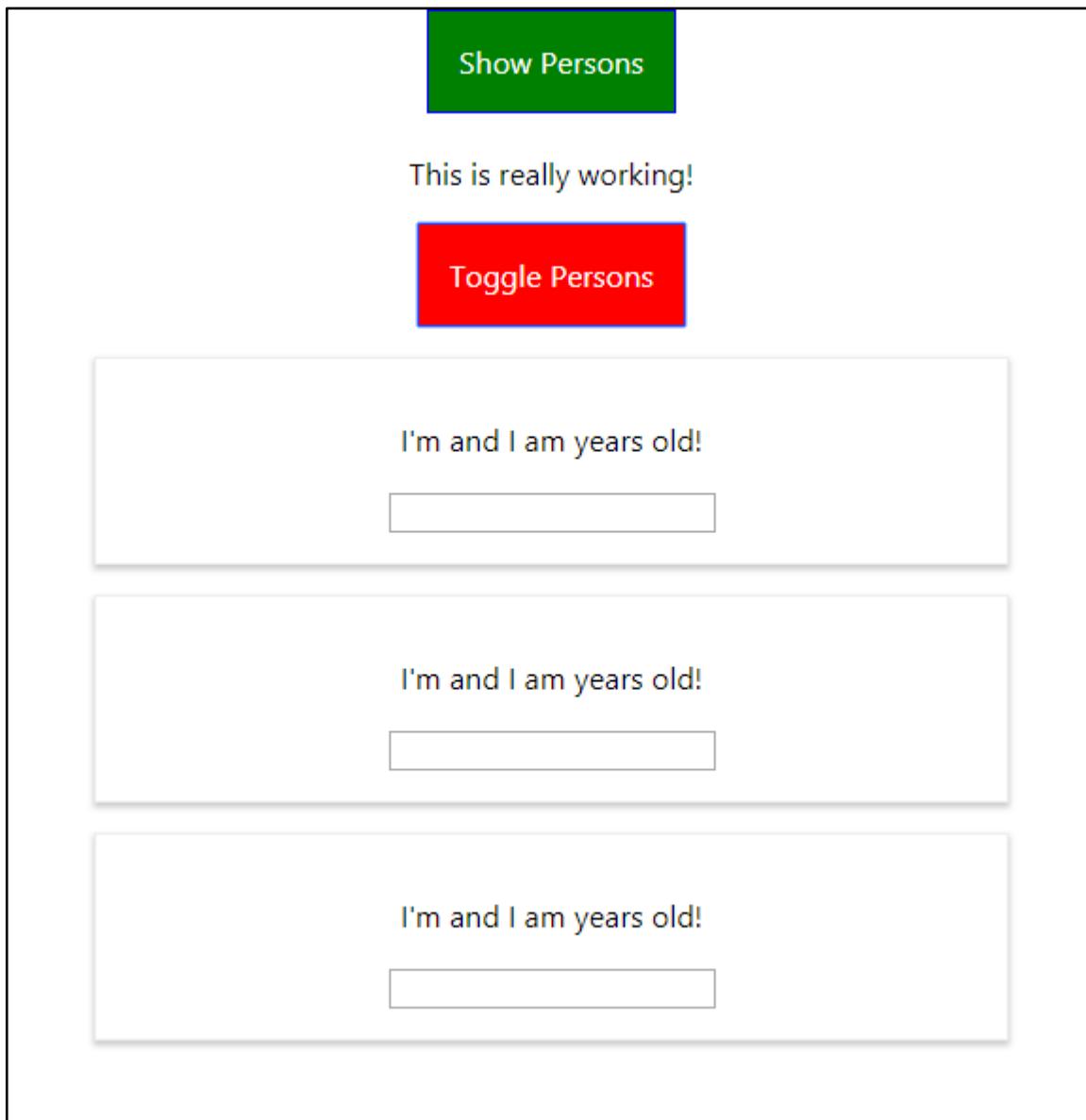
export default withRouter(App, classes.App);
```

Person.js

```
render() {
  console.log("[Person.js] Inside render()");
  return (
    < 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>
  );
}
}

export default withRouter(Person, classes.Person);
```

O/P:



In the above screenshot, all the dynamic elements (props on Person) are missing.

7.20. Passing Unknown Props

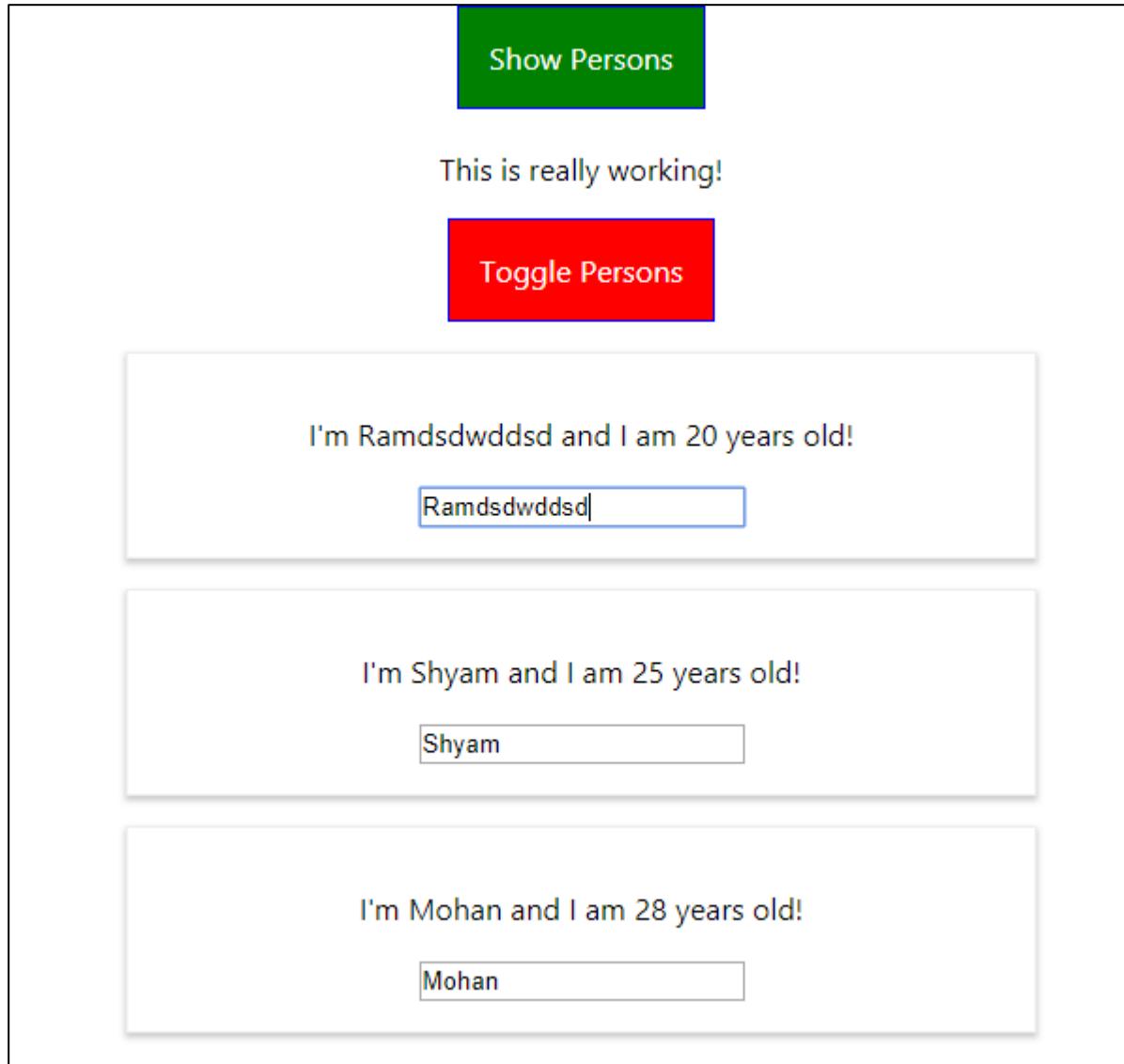
The reason behind missing props in the previous lecture is that WrappedComponent can be Person.js or App.js. We are passing any props in WrappedComponent. Use ES6 feature (Spread operator) to pass props. The below js is now truly reusable higher order component.

withClass.js

```
import React from 'react';
//Takes a configuration and returns a function
const withClass = (WrappedComponent, className) => {
  return (props) => (<div className={className}>
    {/* Never manipulate your WrappedComponent here, just use it */}
  
```

```
/* Pass the props as you get them */
<WrappedComponent {...props} />
</div>
)
}
export default withRouter;
```

O/P:



This higher order component does not have to return a higher order component. If you need access to lifecycle hooks or want to reach out to web, you will need a Stateful component.

withClass.js

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

```

//Takes a configuration and returns a function
// const withClass = (WrappedComponent, className) => {
//     return (props) => (<div className={className}>
//         /* Never manipulate your WrappedComponent here, just use it */
//         /* Pass the props as you get them */
//         <WrappedComponent {...props} />
//     </div>
//     )
// }

const withClass = (WrappedComponent, className) => {
    // Returning an anonymous class
    return class extends Component {
        render() {
            return (
                <div className={className}>
                    <WrappedComponent {...this.props} />
                </div>
            )
        }
    }
}
export default withClass;

```

7.21. Using setState correctly

- 1) Use setState() in an immutable way
- 2) Changing states -> Want to know how many times 'Toggle Persons' is clicked.

App.js

```

togglePersonsHandler = () => {
    const doesState = this.state.showPersons;
    // This will merge 'showPersons' attribute with state object
    this.setState({ showPersons: !doesState, toggleClicked:
this.state.toggleClicked + 1 });
};

```

This is not the correct way of doing it. As setState() call is asynchronous, it does not guarantee the expected result. **Other setState () call might finish before this one. If your new state depends on the**

old state and there is a danger of editing the old state at the same point of time anywhere else in the application. Correct way:

App.js

```
togglePersonsHandler = () => {
  const doesState = this.state.showPersons;
  this.setState((prevState, props) => {
    return {
      showPersons: !doesState,
      toggleClicked: this.state.toggleClicked + 1
    }
  });
};
```

7.22. Validating Props

Add a package (prop-types) to validate props.

```
E:\react-workspace\my-react-app> npm install --save prop-types
```

Person.js

```
import React, { Component } from "react";
import withClass from "../../../../../hoc/withClass";
import classes from "../../../../../components/Persons/Person/Person.css";
import Aux from "../../../../../hoc/Aux";
import PropTypes from 'prop-types';

class Person extends Component {
  constructor(props) {
    super(props);
    console.log("[Person.js] Inside Constructor", props);
  }
  componentWillMount() {
    console.log("[Person.js] Inside componentWillMount()");
  }
  componentDidMount() {
    console.log("[Person.js] Inside componentDidMount()");
  }

  render() {
    console.log("[Person.js] Inside render()");
  }
}
```

```

    return (
      < 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>
    );
  }
}

Person.propTypes = {
  click: PropTypes.func,
  name: PropTypes.string,
  age: PropTypes.number,
  changed: PropTypes.func
}
export default withRouter(Person);

```

'prop-types' does not work in functional components.

7.23. Available PropTypes

Source: <https://reactjs.org/docs/typechecking-with-proptypes.html>

```

import PropTypes from 'prop-types';

MyComponent.propTypes = {
  // You can declare that a prop is a specific JS primitive. By default, these
  // are all optional.
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
  optionalSymbol: PropTypes.symbol,

  // Anything that can be rendered: numbers, strings, elements or an array
  // (or fragment) containing these types.
  optionalNode: PropTypes.node,
}

```

```

// A React element.
optionalElement: PropTypes.element,

// You can also declare that a prop is an instance of a class. This uses
// JS's instanceof operator.
optionalMessage: PropTypes.instanceOf(Message),

// You can ensure that your prop is limited to specific values by treating
// it as an enum.
optionalEnum: PropTypes.oneOf(['News', 'Photos']),

// An object that could be one of many types
optionalUnion: PropTypes.oneOfType([
  PropTypes.string,
  PropTypes.number,
  PropTypes.instanceOf(Message)
]),

// An array of a certain type
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

// An object with property values of a certain type
optionalObjectOf: PropTypes.objectOf(PropTypes.number),

// An object taking on a particular shape
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),

// You can chain any of the above with `isRequired` to make sure a warning
// is shown if the prop isn't provided.
requiredFunc: PropTypes.func.isRequired,

// A value of any data type
requiredAny: PropTypes.any.isRequired,

// You can also specify a custom validator. It should return an Error
// object if the validation fails. Don't `console.warn` or throw, as this
// won't work inside `oneOfType`.
customProp: function(props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error(
      'Invalid prop `' + propName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
  }
},

// You can also supply a custom validator to `arrayOf` and `objectOf`.
// It should return an Error object if the validation fails. The validator
// will be called for each key in the array or object. The first two
// arguments of the validator are the array or object itself, and the
// current item's key.
customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName,
location, propFullName) {
  if (!/matchme/.test(propValue[key])) {
    return new Error(
      'Invalid prop `' + propFullName + '` supplied to' +

```

```

        ' ` ` + componentName + ``. Validation failed.'
    );
}
})
};


```

Requiring Single Child

With `PropTypes.element` you can specify that only a single child can be passed to a component as `children`.

```

import PropTypes from 'prop-types';

class MyComponent extends React.Component {
  render() {
    // This must be exactly one element or it will warn.
    const children = this.props.children;
    return (
      <div>
        {children}
      </div>
    );
  }
}

MyComponent.propTypes = {
  children: PropTypes.element.isRequired
};


```

Default Prop Values

You can define default values for your `props` by assigning to the special `defaultProps` property:

```

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

// Specifies the default values for props:
Greeting.defaultProps = {
  name: 'Stranger'
};

// Renders "Hello, Stranger":
ReactDOM.render(
  <Greeting />,
  document.getElementById('example')
);


```

The `defaultProps` will be used to ensure that `this.props.name` will have a value if it was not specified by the parent component. The `propTypes` typechecking happens after `defaultProps` are resolved, so typechecking will also apply to the `defaultProps`.

7.24. Using References (ref)

Ref takes a dynamic input which should be a function. References are only available in Stateful components.

Focus the text input of the first person.

'ref' takes a dynamic input which should be a function. In the arrow function:

- 1) Set up reference to this element (`<input>`)
- 2) `thisInputElement` will create a new property for the entire class which you can use anywhere in the class. This property will give access to the `<input>` element.

You can also use 'ref' on your own component, not only HTML elements. **DO NOT change styles with the help of 'ref'.**

'ref' is used for few selected things: → controlling focus, media playback

Person.js

```
import React, { Component } from "react";
import withClass from "../../hoc/withClass";
import classes from "../../components/Persons/Person/Person.css";
import Aux from "../../hoc/Aux";
import PropTypes from 'prop-types';

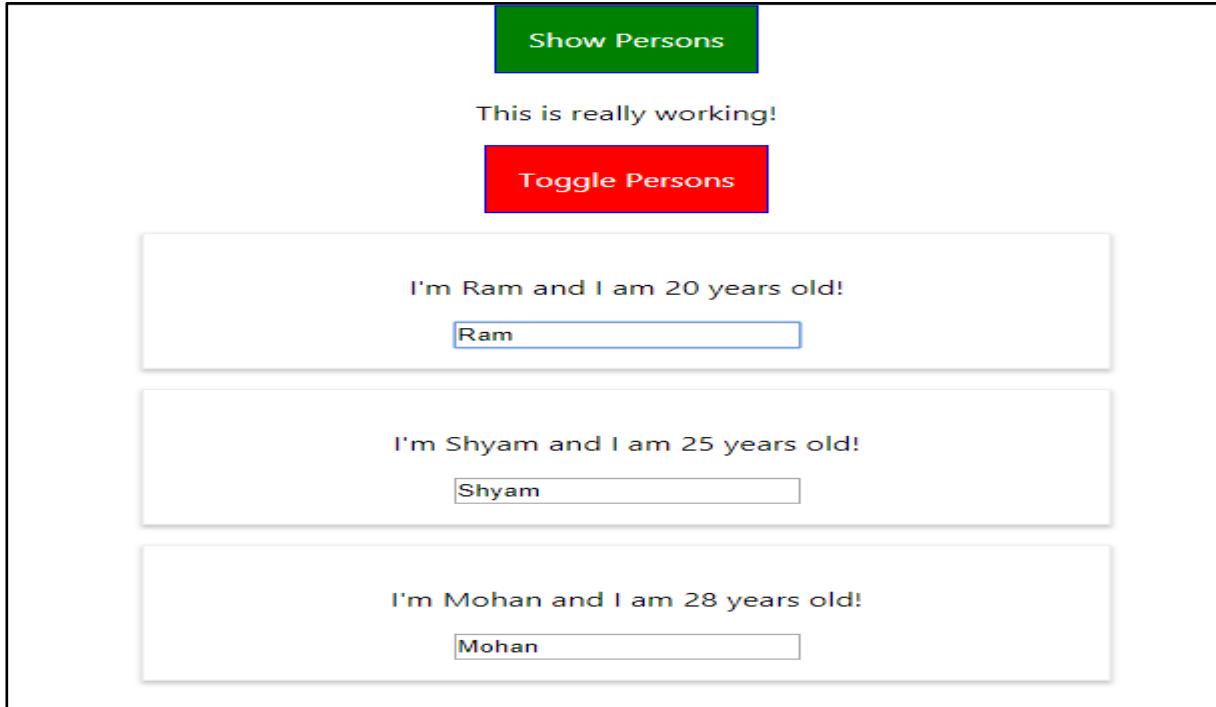
class Person extends Component {
  constructor(props) {
    super(props);
    console.log("[Person.js] Inside Constructor", props);
  }
  componentWillMount() {
    console.log("[Person.js] Inside componentWillMount()");
  }
  componentDidMount() {
    console.log('[Person.js] Inside componentDidMount()');
    if (this.props.position === 0) {
      this.inputElement.focus();
    }
  }
  render() {
```

```

console.log("[Person.js] Inside render()");
return (
  <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
      ref={(inp) => { this.inputElement = inp }}
      type="text"
      onChange={this.props.changed}
      value={this.props.name} />
  </Aux>
)
}
Person.propTypes = {
  click: PropTypes.func,
  name: PropTypes.string,
  age: PropTypes.number,
  changed: PropTypes.func
}
export default withRouter(Person);

```

O/P:



The lectures with [LEGACY] are features from react 16.3 to 16.6.

57. [LEGACY] More on the React ref API (16.3)

In react 16.3 we got a more convenient way of rendering 'ref'.

Feature1:

Person.js

```
import React, { Component } from "react";
import withClass from "../../hoc/withClass";
import classes from "../../components/Persons/Person/Person.css";
import Aux from "../../hoc/Aux";
import PropTypes from 'prop-types';

class Person extends Component {
  constructor(props) {
    super(props);
    console.log("[Person.js] Inside Constructor", props);
    this.inputElement = React.createRef();
  }

  componentWillMount() {
    console.log("[Person.js] Inside componentWillMount()");
  }

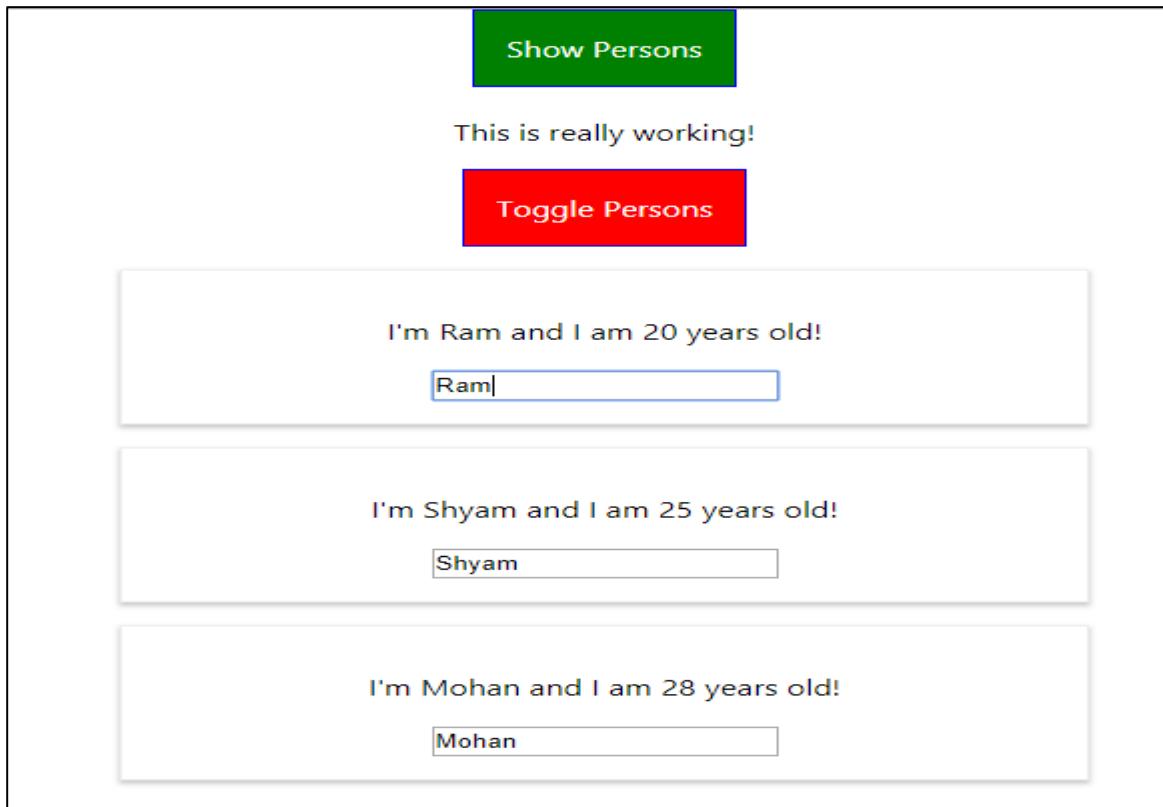
  componentDidMount() {
    console.log('[Person.js] Inside componentDidMount()');
    if (this.props.position === 0) {
      // this.inputElement.focus();
      // current holds the reference to the element we assign the
      // reference to
      this.inputElement.current.focus();
    }
  }
  render() {
    console.log("[Person.js] Inside render()");
    return (
      <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
  // ref={(inp) => { this.inputElement = inp }}
  ref={this.inputElement}
  type="text"
  onChange={this.props.changed}
  value={this.props.name} />
</Aux>
)
}
}
Person.propTypes = {
  click: PropTypes.func,
  name: PropTypes.string,
  age: PropTypes.number,
  changed: PropTypes.func
}
export default withRouter(Person, classes.Person);

```

O/P (When u click on 'toggle persons' , you see first input selected but this time using ref.



Feature2: Let us say we have a new method in Person.js.

- Add a method (focusInput()) in Person.js
- Move componentDidMount() code into focusInput() method
- Let us say you want to call this focusInput() method from Persons.js (ie from outside the component)
- Create ref in <person> in Persons.js
- In componentDidMount() write code to focus

Person.js changes:

- Change focusInput() to focus()
- This focus() method is called from outside (person.js)

forwardRef →

Whatever we are passing as props in <Person> in Persons.js, is received as props in the below code in **withClass.js**.

```
// forwardRef is the higher order component which we export in this
file and which we then use in Person.js file
    // forwardRef is a method provided by react which in the end will
simply get the props you passed to your component
    return React.forwardRef((props, ref) => {
        // You can give any name in place of 'forwardedRef'
        return <withClass {...props} forwardedRef={ref} />
    });
}
```

Ref (2nd arg) is any reference we are using. 'ref' refers to the reference passed to <person> in persons.js.

What we are essentially doing with this chain of ref's is that we are creating a direct tunnel between person.js and its parent component (Persons.js) and that tunnel will ignore any higher order component that are in between, thanks to forwardRef call in withClass.js.

You need forwardRef in the cases where you have wrapping higher order component and you want to create a tunnel for your references so that you directly get access to the underlying wrapped component.

Persons.js

```
import React, { PureComponent } from "react";
import Person from "./Person/Person";

class Persons extends PureComponent {
    constructor(props) {
```

```
super(props);
console.log("[Persons.js] Inside Constructor", props);
this.lastPersonRef = React.createRef();
}

componentWillMount() {
  console.log("[Persons.js] Inside componentWillMount()");
}

componentDidMount() {
  console.log("[Persons.js] Inside componentDidMount()");
  this.lastPersonRef.current.focus();
}

componentWillReceiveProps(nextProps) {
  console.log(
    "[UPDATE Persons.js] Inside componentWillReceiveProps",
    nextProps
  );
}

// shouldComponentUpdate(nextProps, nextState) {
//   console.log(
//     "[UPDATE Persons.js] Inside shouldComponentUpdate",
//     nextProps,
//     nextState
//   );
//   return nextProps.persons !== this.props.persons ||
//     nextProps.changed !== this.props.changed ||
//     nextProps.clicked !== this.props.clicked;
//   // return true;
// }

componentWillUpdate(nextProps, nextState) {
  console.log(
    "[UPDATE Persons.js] Inside componentWillUpdate",
    nextProps,
    nextState
  );
}
```

```

componentDidUpdate() {
  console.log("[UPDATE Persons.js] Inside componentDidUpdate");
}

render() {
  console.log("[Persons.js] Inside render()");
  return this.props.persons.map((person, index) => {
    return (
      <Person
        click={() => this.props.clicked(index)}
        name={person.name}
        age={person.age}
        key={person.id}
        ref={this.lastPersonRef}
        changed={event => this.props.changed(event, person.id)}
        position={index}
      />
    );
  });
}

export default Persons;

```

Person.js

```

import React, { Component } from "react";
import withClass from "../../hoc/withClass";
import classes from "../../components/Persons/Person/Person.css";
import Auxi from "../../hoc/Auxi";
import PropTypes from 'prop-types';

class Person extends Component {
  constructor(props) {
    super(props);
    console.log("[Person.js] Inside Constructor", props);
    this.inputElement = React.createRef();
  }
}

export default withClass(Person, classes);

```

```

}

componentWillMount() {
  console.log("[Person.js] Inside componentWillMount()");
}

componentDidMount() {
  console.log('[Person.js] Inside componentDidMount()');
  if (this.props.position === 0) {
    // this.inputElement.focus();
    // current holds the reference to the element we assign the
    reference to
    this.inputElement.current.focus();
  }
}

focus() {
  this.inputElement.current.focus();
}

render() {
  console.log("[Person.js] Inside render()");
  return (
    <Auxi>
      <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
        // ref={(inp) => { this.inputElement = inp }}
        ref={this.inputElement}
        type="text"
        onChange={this.props.changed}
        value={this.props.name} />
    </Auxi>
  )
}
}

Person.propTypes = {
  click: PropTypes.func,
}

```

```

        name: PropTypes.string,
        age: PropTypes.number,
        changed: PropTypes.func
    }

// Now withClass refers to forwardRef method result
export default withClass(Person, classes.Person);

```

wihClass.js

```

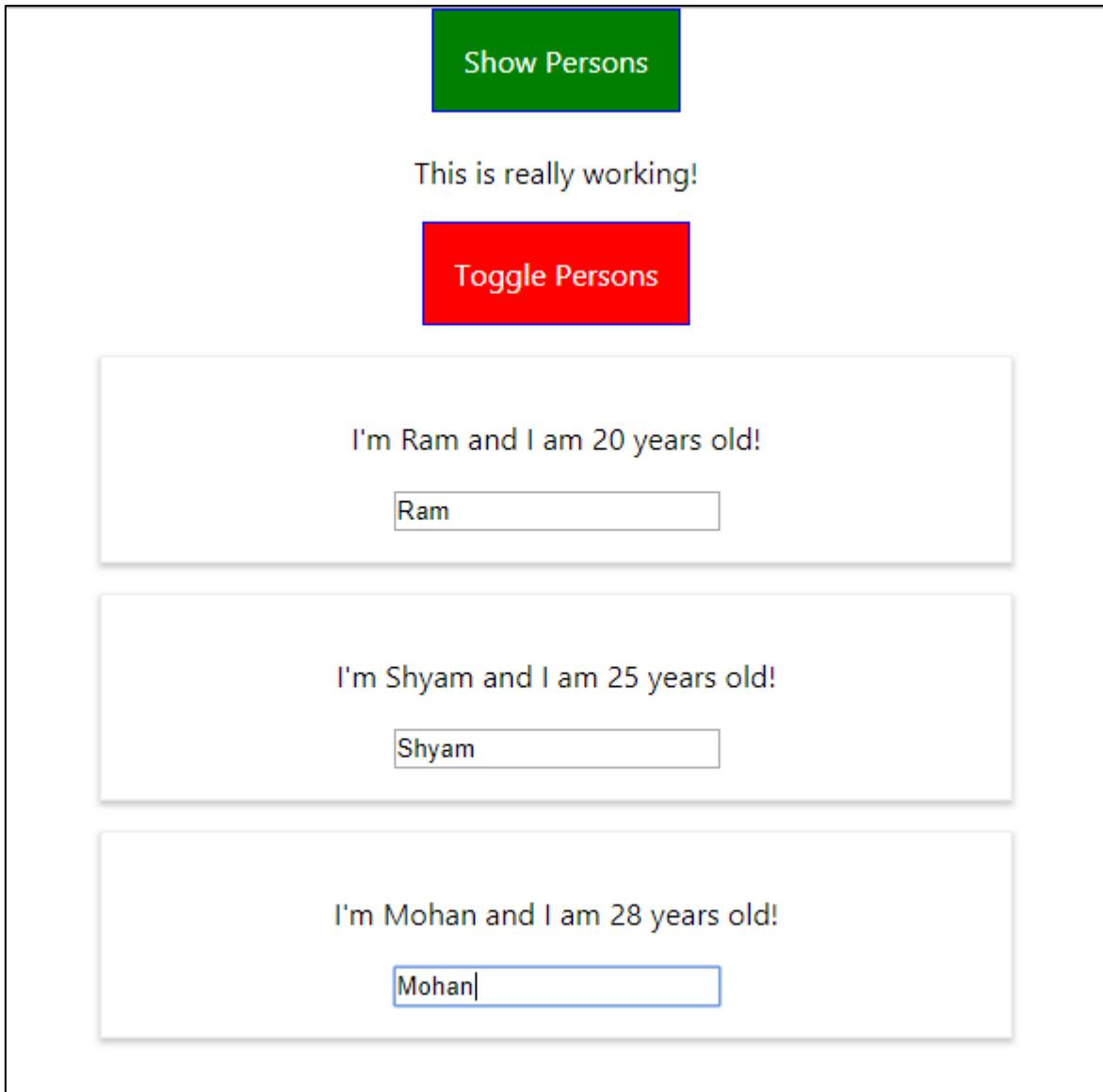
import React, { Component } from 'react';
//Takes a configuration and returns a function
// const withClass = (WrappedComponent, className) => {
//     return (props) => (<div className={className}>
//         {/* Never manipulate your WrappedComponent here, just use
// it */}
//         {/* Pass the props as you get them */}
//         <WrappedComponent {...props} />
//     </div>
//     )
// }

const withClass = (WrappedComponent, className) => {
    const WithClass = class extends Component {
        render() {
            return (
                <div className={className}>
                    <WrappedComponent ref={this.props.forwardedRef}
{...this.props} />
                </div>
            )
        }
    }
    // forwardRef is the higher order component which we export in
    // this file and which we then use in Person.js file
    // forwardRef is a method provided by react which in the end will
    // simply get the props you passed to your component
    return React.forwardRef((props, ref) => {

```

```
// You can give any name in place of 'forwardedRef'  
return <WithClass {...props} forwardedRef={ref} />  
});  
  
export default withClass;
```

O/P:



Note: Why is the last element selected unlike before? Think.....

58. [LEGACY] The Context API (React 16.3)

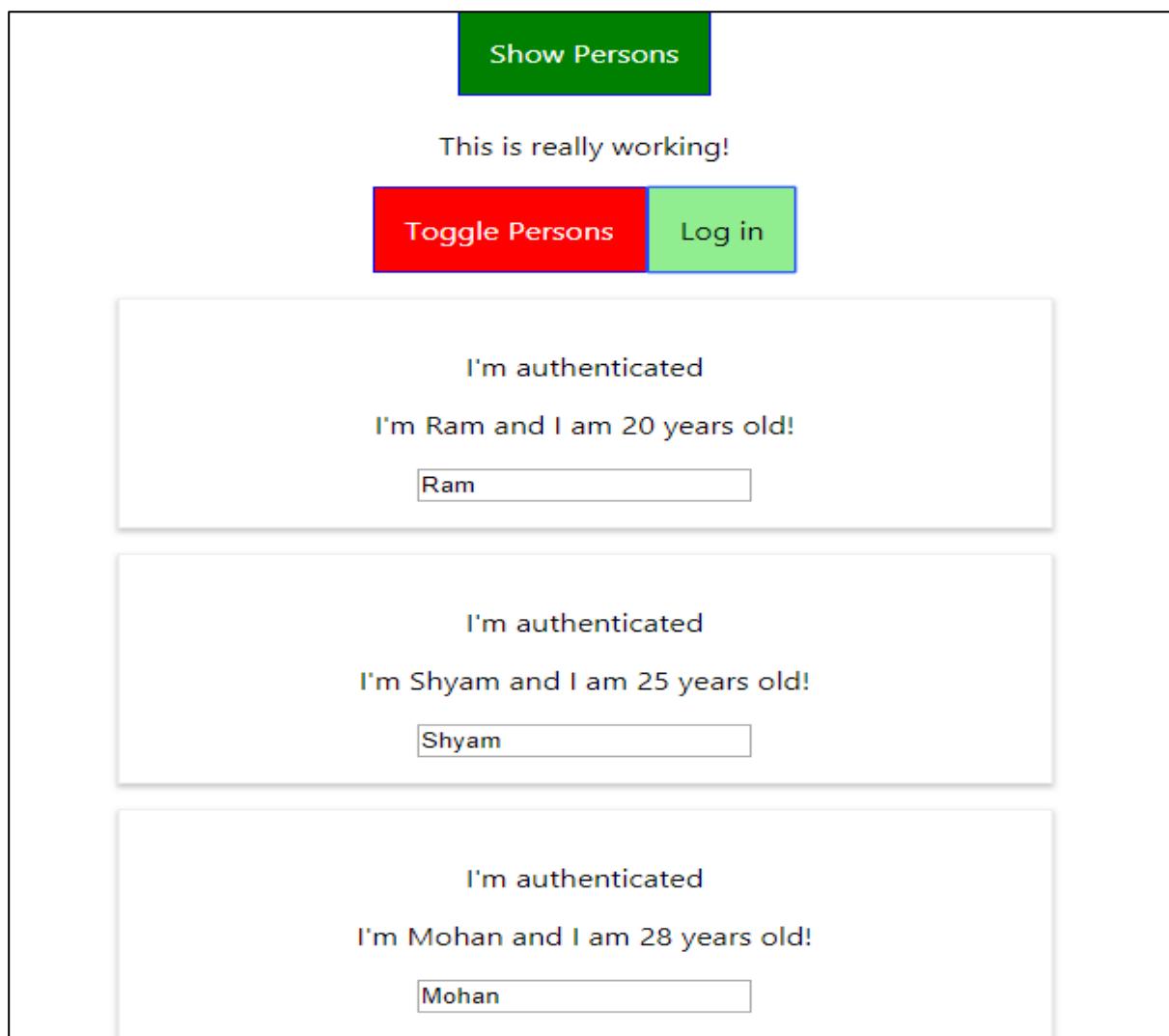
A way to pass global state around in your app.

Use cases: Authentication, global theme color

- Add a button in Cockpit.js. This button once clicked should set some global login state to true and change something on different components.
- Go to App.js and add 'login' property to <Cockpit> and add a loginHandler method to set authenticated state to true.
- Let us say we want to change something in person.js when authenticated property is set to true.
- Pass isAuthenticated property to Persons.js
- Output text in person.js when a person is authenticated.

The default way to achieve above is using props. (See git commit).

O/P (After clicking 'Log in'):



As the authentication is a global state, we can use React's context API in place of passing props to achieve the same result as above.

Steps:

In App.js :

- 1) Create context in App.js. Context creation is done outside the class.

```
// false is the default value for the context
// AuthContext is a component (createContext creates a component)
export const AuthContext = React.createContext(false);
```

- 2) Context works with providers and consumers. App.js is the file from where we provide the context. We have to wrap all the components (in App.js JSX code) with **<AuthContext.Provider>** (Provider is a subcomponent created by React) where we plan to use the context.

```
<AuthContext.Provider>{persons}</AuthContext.Provider>
```

- 3) We can also pass value in the provider.

```
<AuthContext.Provider
value={this.state.authenticated}>{persons}</AuthContext.Provider>
```

In Person.js

- 1) Import AuthContext.

```
import { AuthContext } from "../../containers/App";
```

- 2) Now wrap the elements with **<AuthContext.Consumer>** where we want to use the context value.

```
<AuthContext.Consumer>
    {this.props.authenticated === true ? <p>I'm
authenticated</p> : null}
</AuthContext.Consumer>
```

- 3) Now in the above consumer component we can still use the {} but there we need to execute a method which react will execute for us. This method receives one argument and that is the data we are passing down the context.

```
<AuthContext.Consumer>
  {auth => auth ? <p>I'm authenticated</p> : null}
</AuthContext.Consumer>
```

Note: Still the props approach is recommended as this makes your component more reusable and does not tie them together that much. **However context api is good for global settings.**

59. [LEGACY] More on the Context API (16.6)

Using new project.

With react 16.6 new feature it becomes accessing the state in a child component even easier.

Files of starting project using context from React 16.3:

App.js

```
import React, { Component } from 'react';
import Login from './components/Login';
import Profile from './components/Profile';

export const AuthContext = React.createContext({
  isAuth: false,
  // Here we are passing empty function as we are going to override
  // our default context in the Provider
  toggleAuth: () => {}
});

class App extends Component {
  state = {
    isAuth: false
  };

  toggleAuth = () => {
    this.setState(prevState => {
      return {
        isAuth: !prevState.isAuth
      };
    });
  };

  render() {
```

```

        return (
            <AuthContext.Provider
                // Since 'toggleAuth' is included in the context, we
                // can call it even from child components
                value={{ isAuth: this.state.isAuth, toggleAuth:
                this.toggleAuth }}
            >
                <Login />
                <Profile />
            </AuthContext.Provider>
        );
    }
}

export default App;

```

Login.js

```

import React from 'react';
import { AuthContext } from '../App';

const login = props =>
    <AuthContext.Consumer>
        {authContext => {
            return (
                // this will call toggleAuth() [part of the context
                // object] of App.js
                <button onClick={authContext.toggleAuth}>
                    {authContext.isAuthenticated ? 'Logout' : 'Login'}
                </button>
            );
        }}
    </AuthContext.Consumer>
);

export default login;

```

Profile.js

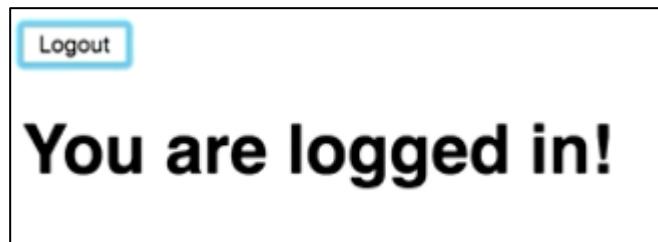
```

import React from 'react';
import { AuthContext } from '../App';

const profile = props => (
  <AuthContext.Consumer>
    {authContext => {
      return (
        <h1>{authContext.isAuthenticated ? 'You are logged in!' : 'Not
logged in!'}</h1>
      );
    }}
  </AuthContext.Consumer>
);
export default profile;

```

O/P:



The above feature is nothing new. This was the old feature. With react **16.6** we got a new feature (**contextType**) that we will use In Login.js. **For this we need to change the login.js from functional component to a class based one.**

Login.js

```

import React, { Component } from 'react';
import AuthContext from '../auth-context';

class Login extends Component {
  // This will tell react which context this component should
  connect to
  // You do not need to use consumer after you add this
  static contextType = AuthContext;

  componentDidMount() {
    console.log(this.context);
  }
}

```

```

}

render() {
    return (
        // this.context is provided by React and it is populated
        // with your context data based on the context type you set up.
        <button onClick={this.context.toggleAuth}>
            {this.context.isAuthenticated ? 'Logout' : 'Login'}
        </button>
    );
}

export default Login;

```

O/P: (Functionality breaks after above change, why?)



Because, the reason is that due to the way data is now being passed around, the old approach of setting up the context in App.js and using it in App.js, importing Login and profile in App.js, and on the other hand importing {AuthContext} from App.js in Login.js, so having like a circular relation, this does not work here. Instead we now have to make one simple change.

- 1) Create auth-context.js and declare AuthContext here (remove it from app.js).
- 2) Import AuthContext from “./auth-context” in App.js
- 3) Change AuthContext import in all the files

With the above 3 changes we now do not have the circular relation anymore. This approach reduces the amount of code we needed to write in Login.js. With the above approach, now you are not only limited to use the context in render method but you can also use it anywhere else in the component (advantage).

Auth-context.js

```

import React from 'react';

export default React.createContext({

```

```
    isAuthenticated: false,
    toggleAuth: () => { }
});
```

App.js

```
import React, { Component } from 'react';
import Login from './components/Login';
import Profile from './components/Profile';
import AuthContext from './auth-context';

class App extends Component {
  state = {
    isAuthenticated: false
  };

  toggleAuth = () => {
    this.setState(prevState => {
      return {
        isAuthenticated: !prevState.isAuthenticated
      };
    });
  };

  render() {
    return (
      <AuthContext.Provider
        value={{ isAuthenticated: this.state.isAuthenticated, toggleAuth: this.toggleAuth }}>
        >
          <Login />
          <Profile />
        </AuthContext.Provider>
      );
    }
}
export default App;
```

60. [LEGACY] Updated Lifecycle Hooks (React 16.3)

Among all the lifecycle methods used in App.js, React **16.3** discourages (however they will continue to work) the use of 3 of them (**componentWillMount**, **componentWillUpdate**, **componentWillReceiveProps**) and introduces 2 new one.

The discouragement is bcz they were used incorrectly. You could call `setState()` in there and do other bad things.

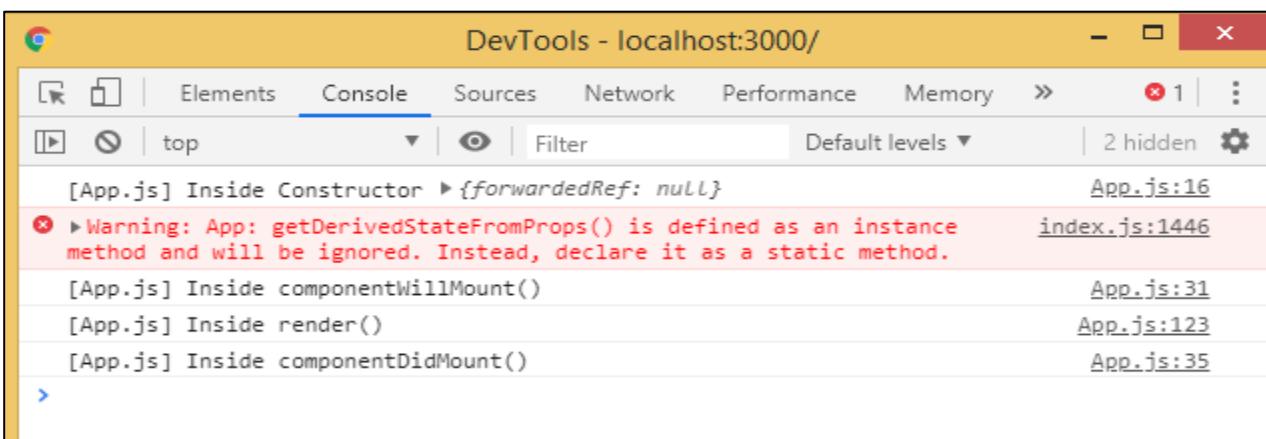
The new lifecycle methods:

- 1) **static getDerivedStateFromProps** → Executed whenever your props are updated. It gives you a chance of updating your state with props. Your state should rarely be coupled to your props. But sometimes you get cases where you receive new props and you want to update your local state bcz maybe you want to work with that state, you want to change it into that component before a user clicks save button. **You should also return a state from this method so that it can be merged with the old state but you can also return the prevState. We also need to use static keyword with this method as it is not attached to a single instance.**

App.js snippet:

```
static getDerivedStateFromProps(nextProps, prevState) {
  console.log(
    "[UPDATE App.js] Inside getDerivedStateFromProps",
    nextProps,
    prevState
  );
  // You must return a valid state object here
  return prevState;
}
```

Screenshot to see the execution order:



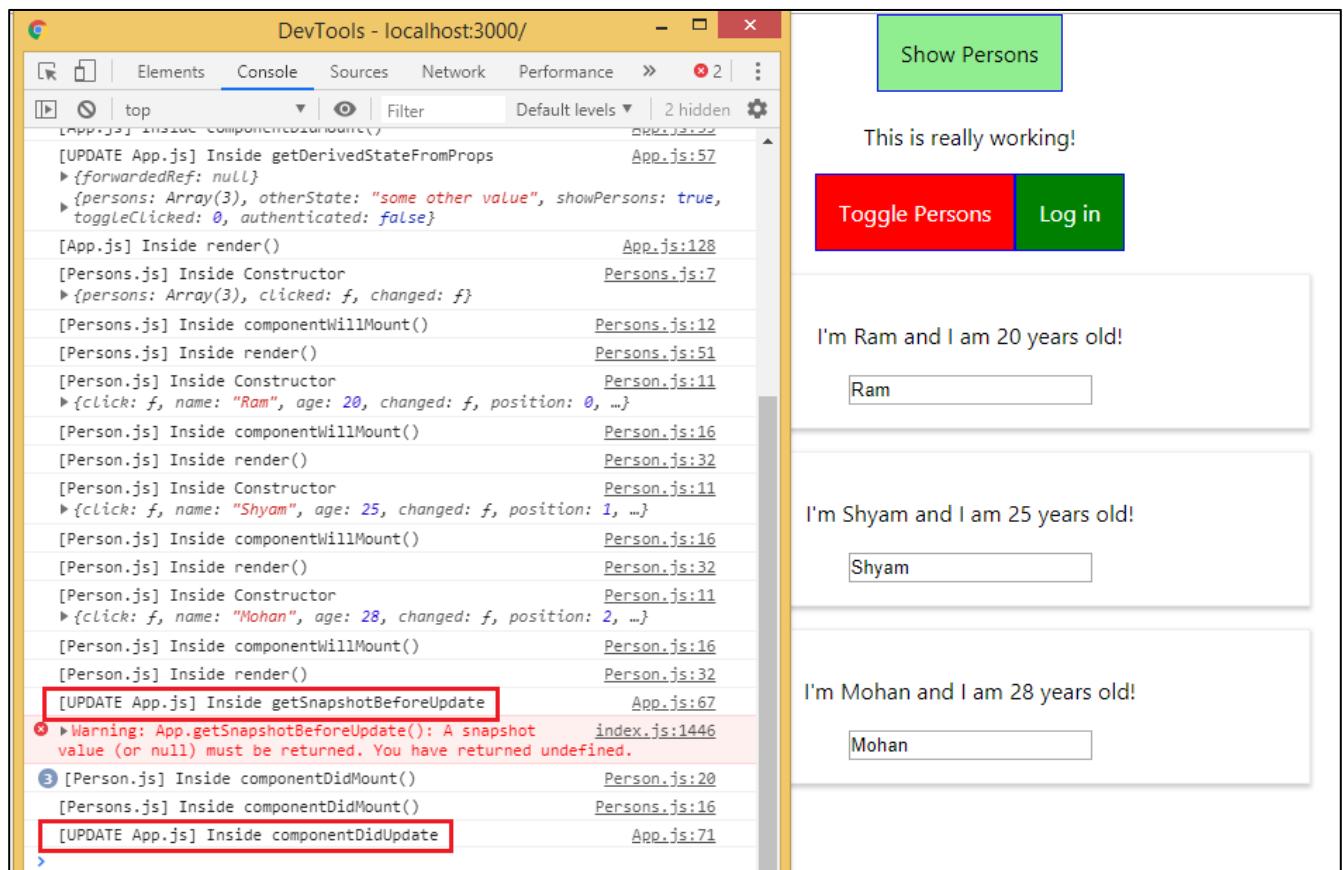
- 2) **static getSnapshotBeforeUpdate** → Allows you to get a snapshot of your DOM right before it is about to change. This runs right before componentDidUpdate() is done for App.js.

Note: In getSnapshotBeforeUpdate you can save before the DOM change and you can scroll the user back to the previously saved position.

App.js snippet:

```
getSnapshotBeforeUpdate() {
  console.log("[UPDATE App.js] Inside getSnapshotBeforeUpdate");
}
```

Screenshot to see the execution order: (After we click 'Show persons')



61. [LEGACY] The memo Method (16.6)

This feature is related to how we can effectively re-render our components. **ShouldComponentUpdate** and **PureComponent** can only be used in class based components. Prior to React 16.6, we always had to convert our component to class based components to take advantage of this optimization. With react 16.6 we can also optimize functional components using the new feature. In React 16.6

there is a new method on the react object itself and you can use them in any functional component. Wrap your component with **React.memo()**.

```
export default React.memo(cockpit);
```

With the above change, the cockpit component will only re-render if the props it receive really change. Shallow check on props will be done. It is good to have this option where the props could be updated with the old value indeed and therefore you might avoid re-rendering. Use it when you think that the child component should not be updated with every change in the parent component.

16. When should you optimize (new lectures)

You should not implement shouldComponentUpdate/React.memo in every component as it comes with the cost of checking extra condition. Only implement them when you think that some changes in parent may not effect child components.

62. [LEGACY] Wrap Up

63. [LEGACY] Useful Resources & Links

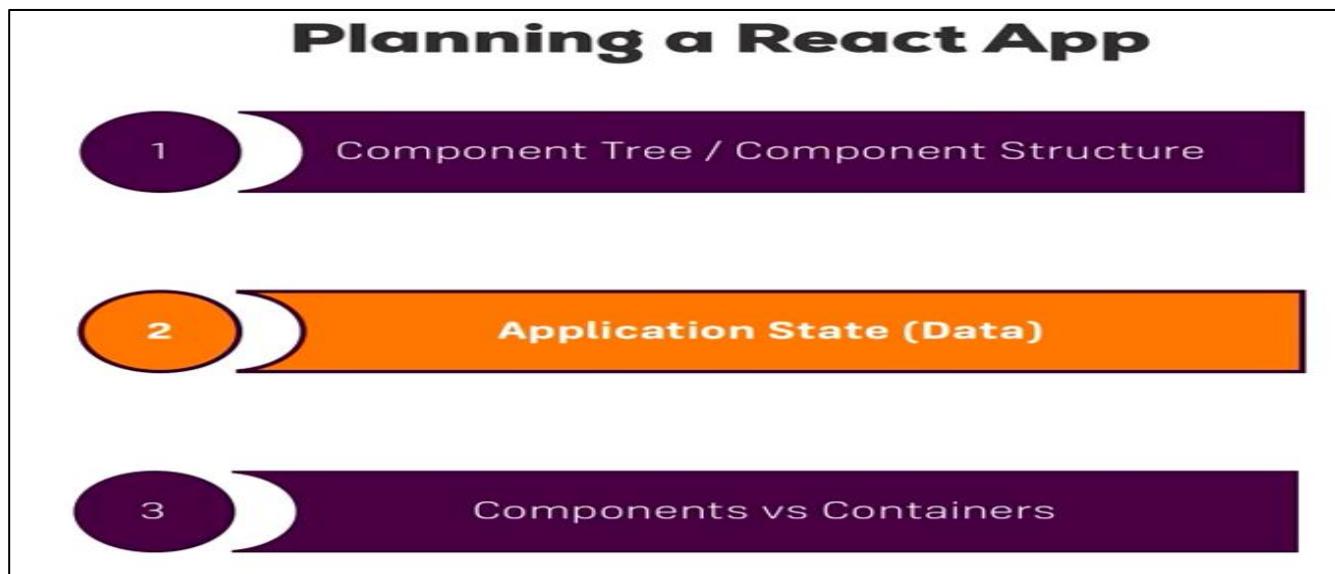
- State & Lifecycle: <https://reactjs.org/docs/state-and-lifecycle.html>
- PropTypes: <https://reactjs.org/docs/typechecking-with-proptypes.html>
- Higher Order Components: <https://reactjs.org/docs/higher-order-components.html>
- Refs: <https://reactjs.org/docs/refs-and-the-dom.html>

8. A Real App The Burger Builder (Basic Version)

8.1. Module Introduction

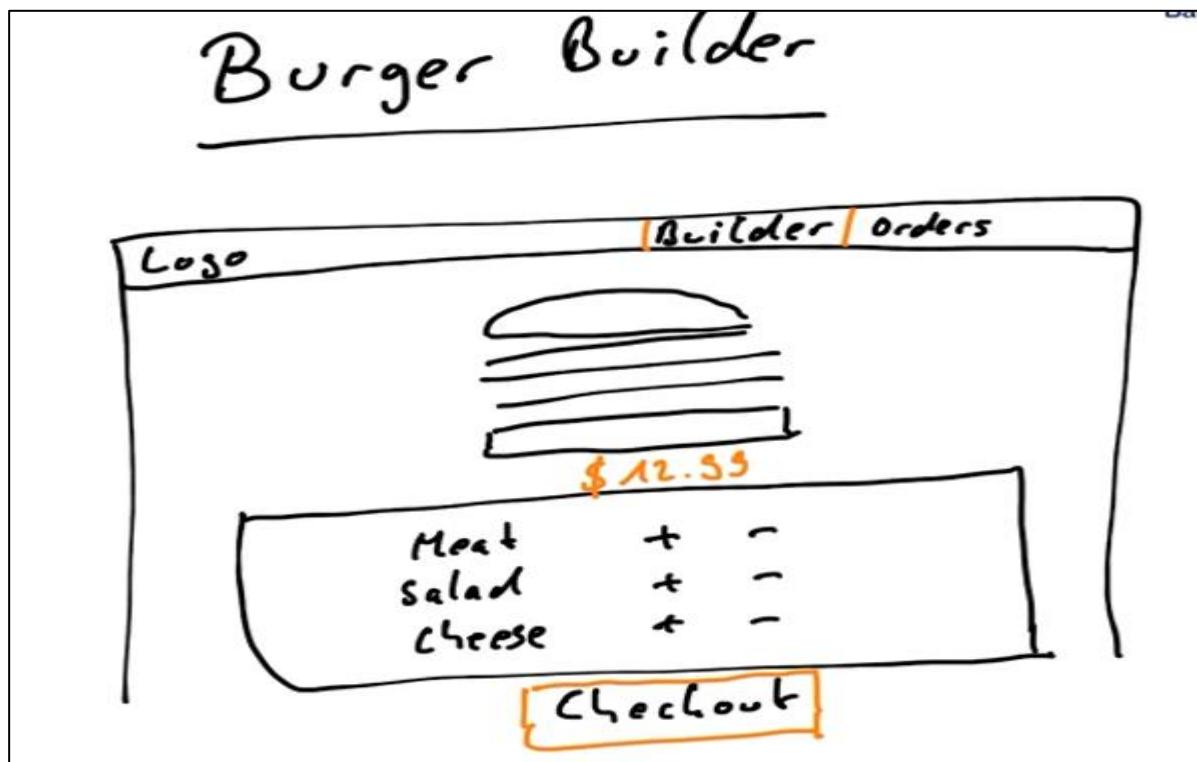
User can dynamically build a burger, add ingredients, and purchase it.

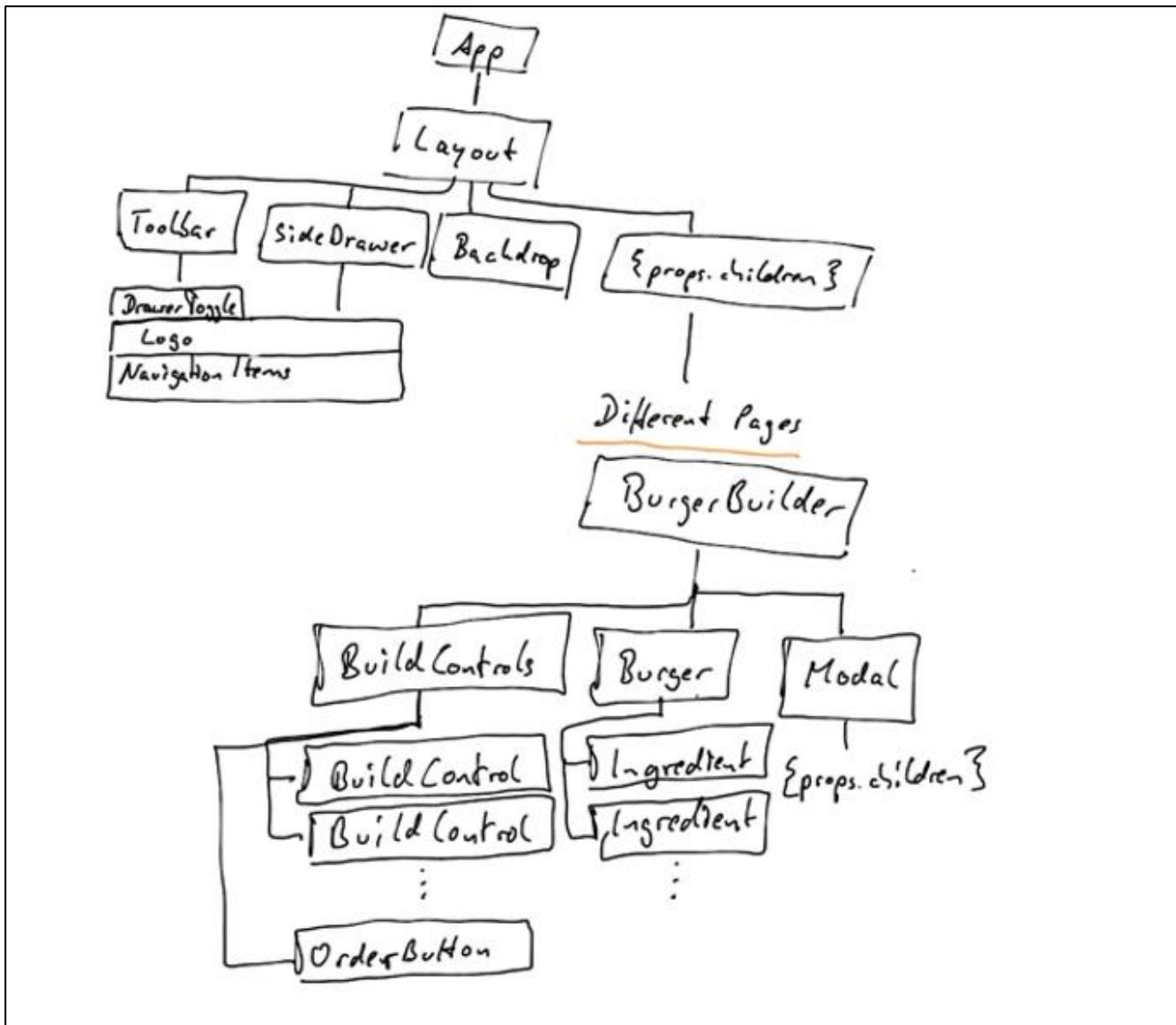
8.2. Planning an App in React - Core Steps



8.3. Planning our App - Layout and Component Tree

Builder page





{props.children} → Can navigate to different pages. For ex, Burger builder page, checkout page. These pages are going to share the same layout. Pass components dynamically depending upon which component we want to view.

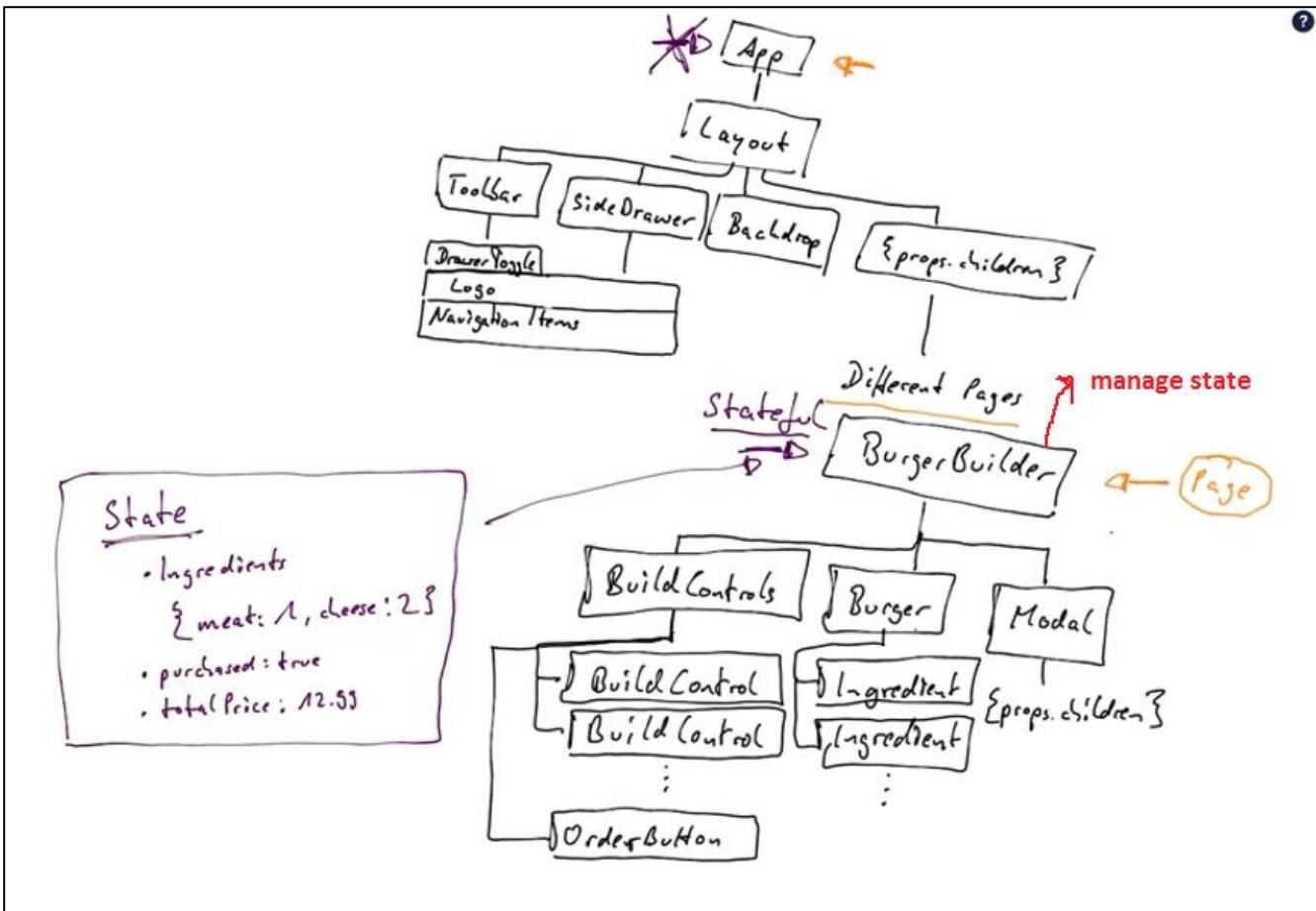
Modal → A wrapper component, which at the end takes {props.children} to wrap itself around any component we want to render in a modal so that we have a reusable modal we could use for all kinds of stuffs like, displaying a confirmation message, error message, order summary.

8.4. Planning the State

This will tell how many Stateful and stateless components we need.

Q) Where should we manage state?

BurgerBuilder.



8.5. Setting up the Project

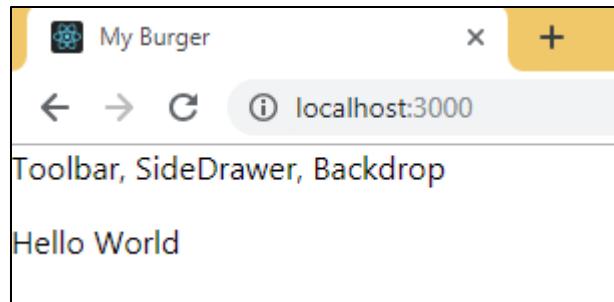
- (1) E:\react-workspace> create-react-app burger
- (2) Enable CssModules (npm run eject). Make changes in webpack.config.js (config folder).
- (3) Delete logo.svg
- (4) Delete <div> content and classname from App.js
- (5) Remove App.css
- (6) Change the font family → go to '<https://fonts.googleapis.com>' and copy cdn for the 'open sans' font and add in index.html.

```
<link href="https://fonts.googleapis.com/css?family=Open+Sans:400,700" rel="stylesheet">
```

8.6. Creating a Layout Component

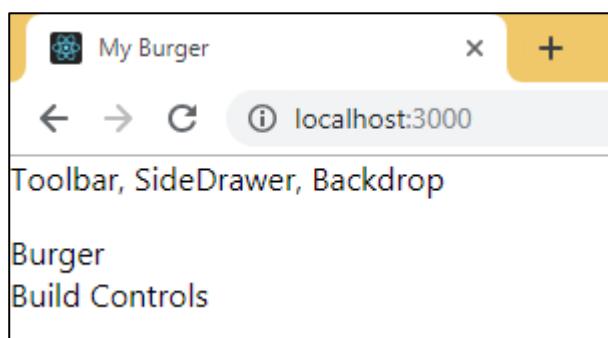
App component can also be a place to wrap entire application, nonetheless creating a separate component Layout.js for the same. This will contain toolbar, sidedrawer, backdrop etc. Below this part we have <main> area where we want to output component we wrap with this layout.

auxiliary.js → Higher order component to wrap Layout component. If you do not want to wrap your component in a root element, you can either return adjacent elements as array elements or use Higher order component to wrap your component.



8.7. Starting Implementation of The Burger Builder Container

Add BurgerBuilder component



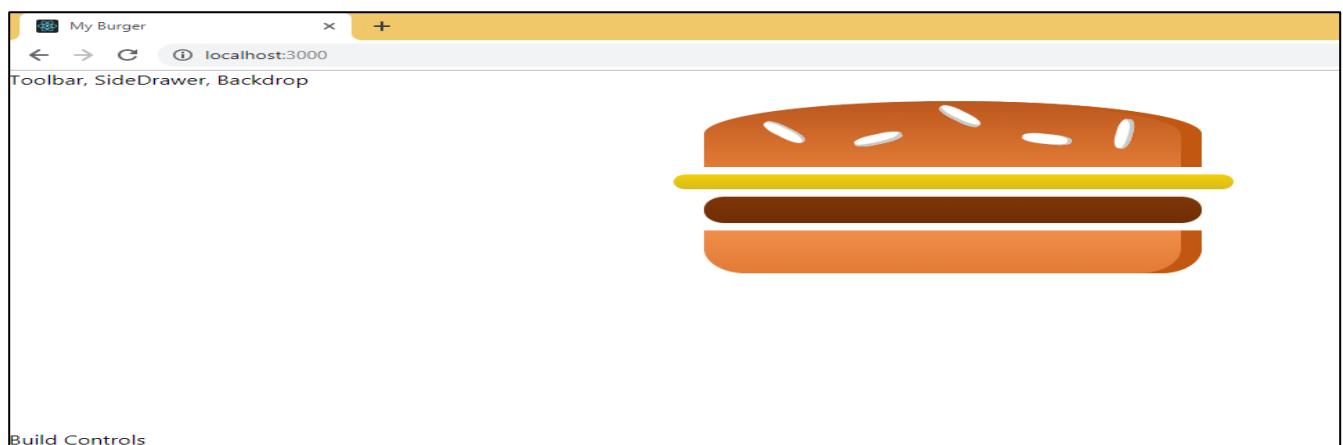
8.8. Adding a Dynamic Ingredient Component

8.9. Adding Prop Type Validation

E:\react-workspace\burger> *npm install --save prop-types*

Prop-types can only be used in class component. Convert BurgerIngredient.js to class component.

8.10. Starting the Burger Component

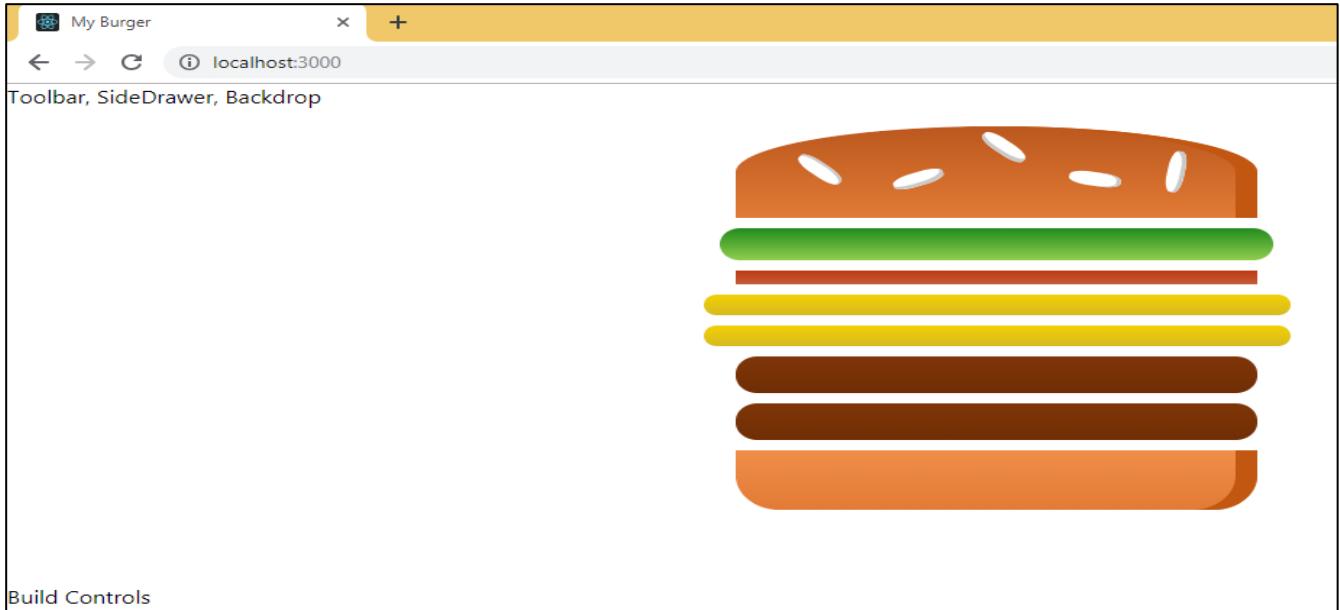


Everything about the above burger is hard-coded. Add ingredients dynamically.

8.11. Outputting Burger Ingredients Dynamically

Object.keys(anyObject) -> Javascript method to get keys of an object

Array(size) -> Javascript method to create undefined array of length 'size'



8.12. Calculating the Ingredient Sum Dynamically

Show message when no ingredients are added.

BurgerBuilder.js

```
import React, { Component } from "react";
import Aux from "../../hoc/auxiliary";
import Burger from "../../Components/Burger/Burger";

class BurgerBuilder extends Component {
  state = {
    ingredients: {
      salad: 0,
      bacon: 0,
      cheese: 0,
      meat: 0
    }
  }

  render() {
```

```

        return (
            <Aux>
                <Burger ingredients={this.state.ingredients}></Burger>
                <div>Build Controls</div>
            </Aux>
        );
    }
}

export default BurgerBuilder;

```

Burger.js

```

import React from 'react';
import classes from './Burger.css';
import BurgerIngredient from './BurgerIngredient/BurgerIngredient';

const burger = (props) => {
    /*
        1. Take keys from 'props.ingredients' object
        2. Create an array (with undefined values) of size equal to value
        of each key
        3. return JSX with key (mandatory bcz we return list of elements
        here) and type
    */
    // Converting object into Array of the value if the ingredients
    // Object.keys(anyObject) -> Javascript method to get keys of an object
    let transformedIngredients = Object.keys(props.ingredients)
        .map(igKey => {
            // Array(size) -> Javascript method to create undefined array
            // of length 'size'
            return [...Array(props.ingredients[igKey])].map((_, i) => {
                return <BurgerIngredient key={igKey + i} type={igKey} />;
            });
        });
    console.log(transformedIngredients);
    return (
        // Wrapping in <div> bcz we want to apply styles here
        <div className={classes.Burger}>
            <BurgerIngredient type="bread-top" />
            {transformedIngredients}
            <BurgerIngredient type="bread-bottom" />
        </div>
    );
}

```

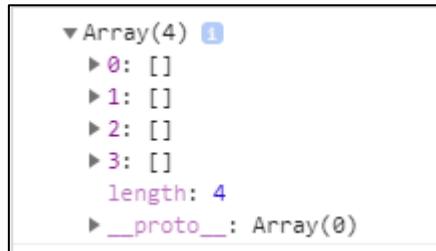
```

        </div>
    );
};

export default burger;

```

console.log In Browser;



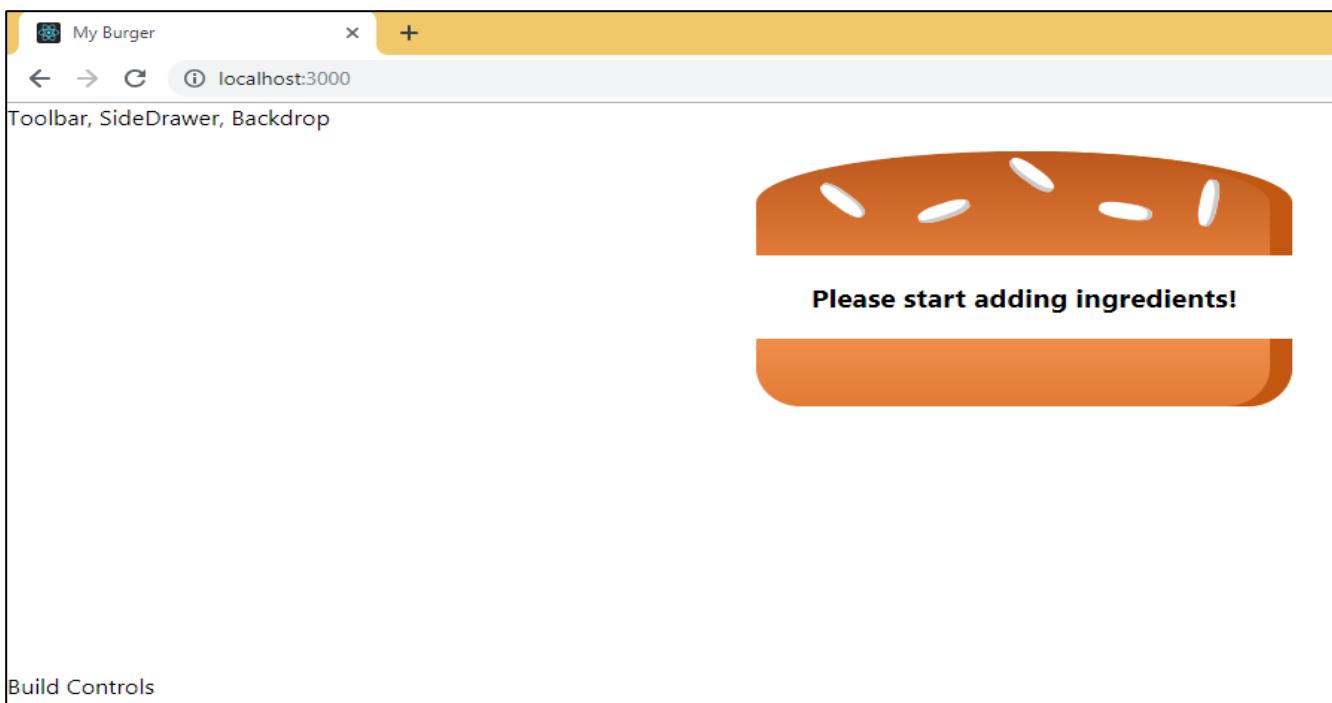
The console above shows 4 empty arrays. So length always shows 4 despite the fact that we do not have any ingredients. So we need to get hold of the inner content of these arrays. In order to achieve this we need to flatten this array to pull out values of these individual array and create one array which contains all these values.

Burger.js

```

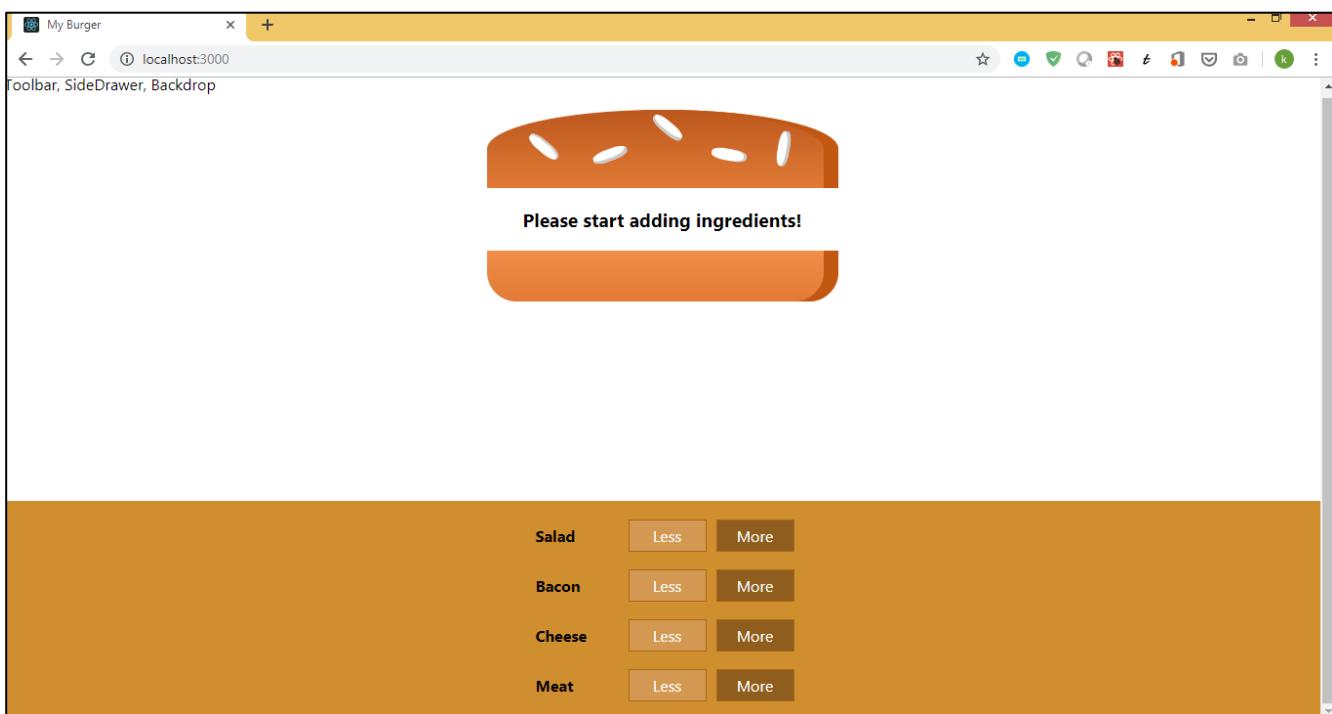
let transformedIngredients = Object.keys(props.ingredients)
    .map(igKey => {
        // Array(size) -> Javascript method to create undefined array
        // of length 'size'
        return [...Array(props.ingredients[igKey])].map(_ , i) => {
            return <BurgerIngredient key={igKey + i} type={igKey} />;
        });
    }).reduce((arr, el) => {
        /*
         * reduce() Flatten the transformedIngredients array. it takes 2
         * arguments,
         * 1. A function
         * 2. Initial reduced value
         */
        return arr.concat(el);
    }, []);
if (transformedIngredients.length === 0) {
    transformedIngredients = <p>Please start adding ingredients!</p>;
}

```



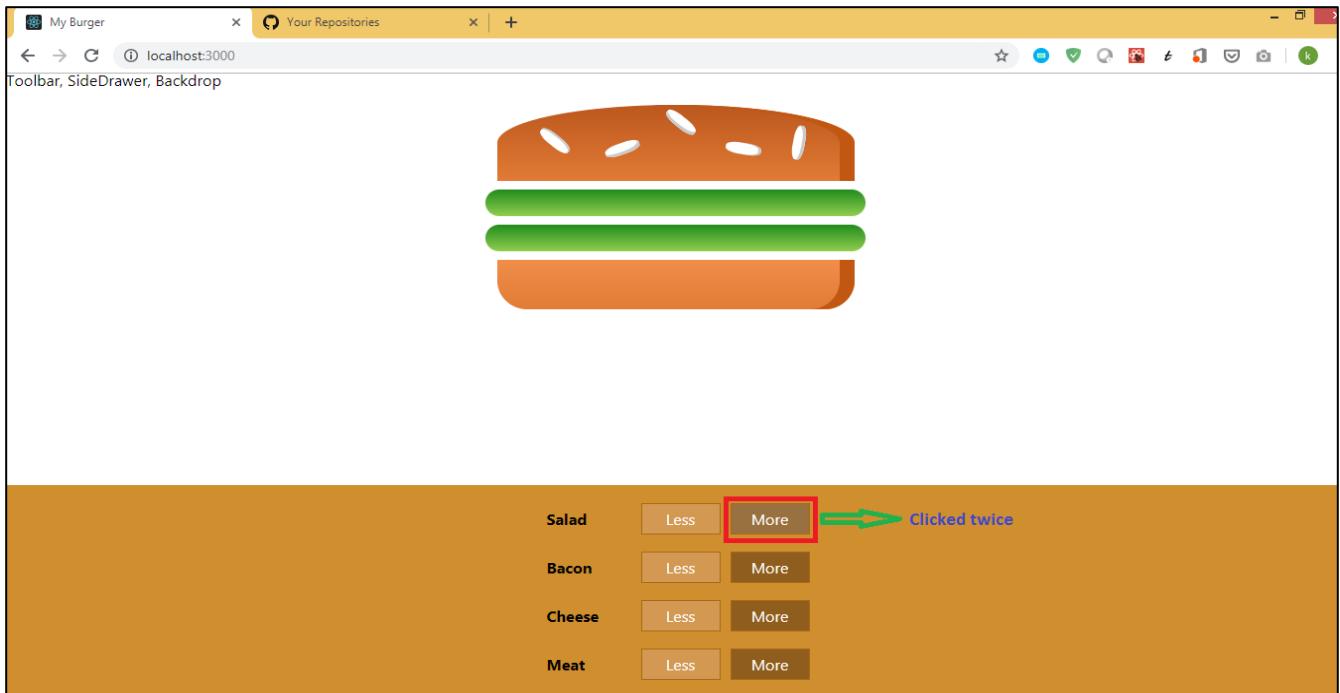
8.13. Adding the Build Control Component

8.14. Outputting Multiple Build Controls



8.15. Connecting State to Build Controls

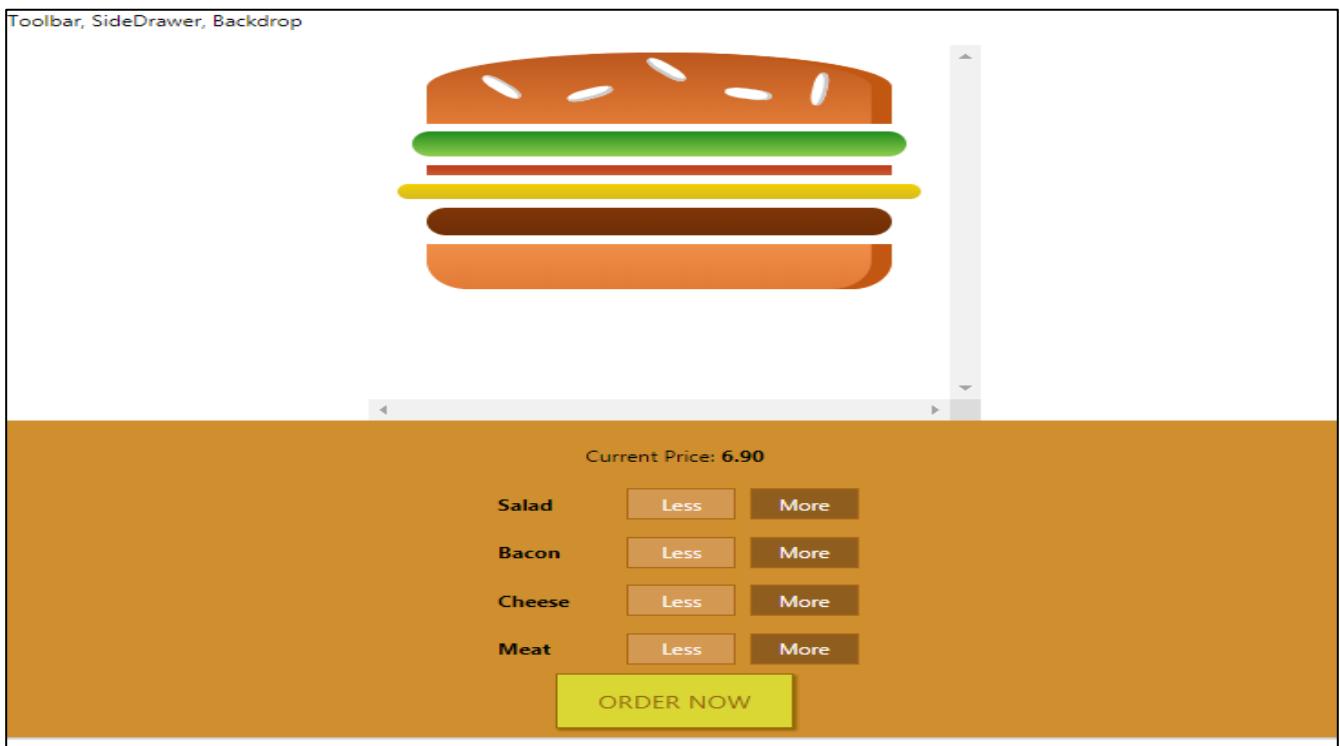
Tie up Buildcontrols to ingredients so that we can manage burger on our own.



8.16. Removing Ingredients Safely

8.17. Displaying and Updating the Burger Price

8.18. Adding the Order Button



8.19. Creating the Order Summary Modal

The idea behind modal is to have a wrapping element which provides the wrapping that wraps around any component you want to show in that model.

Create UI/ Modal/Modal.js.

Create UI/ Backdrop/Backdrop.js.

8.20. Showing & Hiding the Modal (with Animation!)

8.21. Implementing the Backdrop Component

8.22. Adding a Custom Button Component

8.23. Implementing the Button Component

8.24. Adding the Price to the Order Summary

8.25. Adding a Toolbar

8.26. Using a Logo in our Application

8.27. Adding Reusable Navigation Items

8.28. Creating a Responsive Sidedrawer

8.29. Working on Responsive Adjustments

8.30. More about Responsive Adjustments

8.31. Reusing the Backdrop

8.32. Adding a Sidedrawer Toggle Button

8.33 Adding a Hamburger Icon

8.34. Improving the App – Introduction

8.35. Prop Type Validation

8.36. Improving Performance

8.37. Using Component Lifecycle Methods

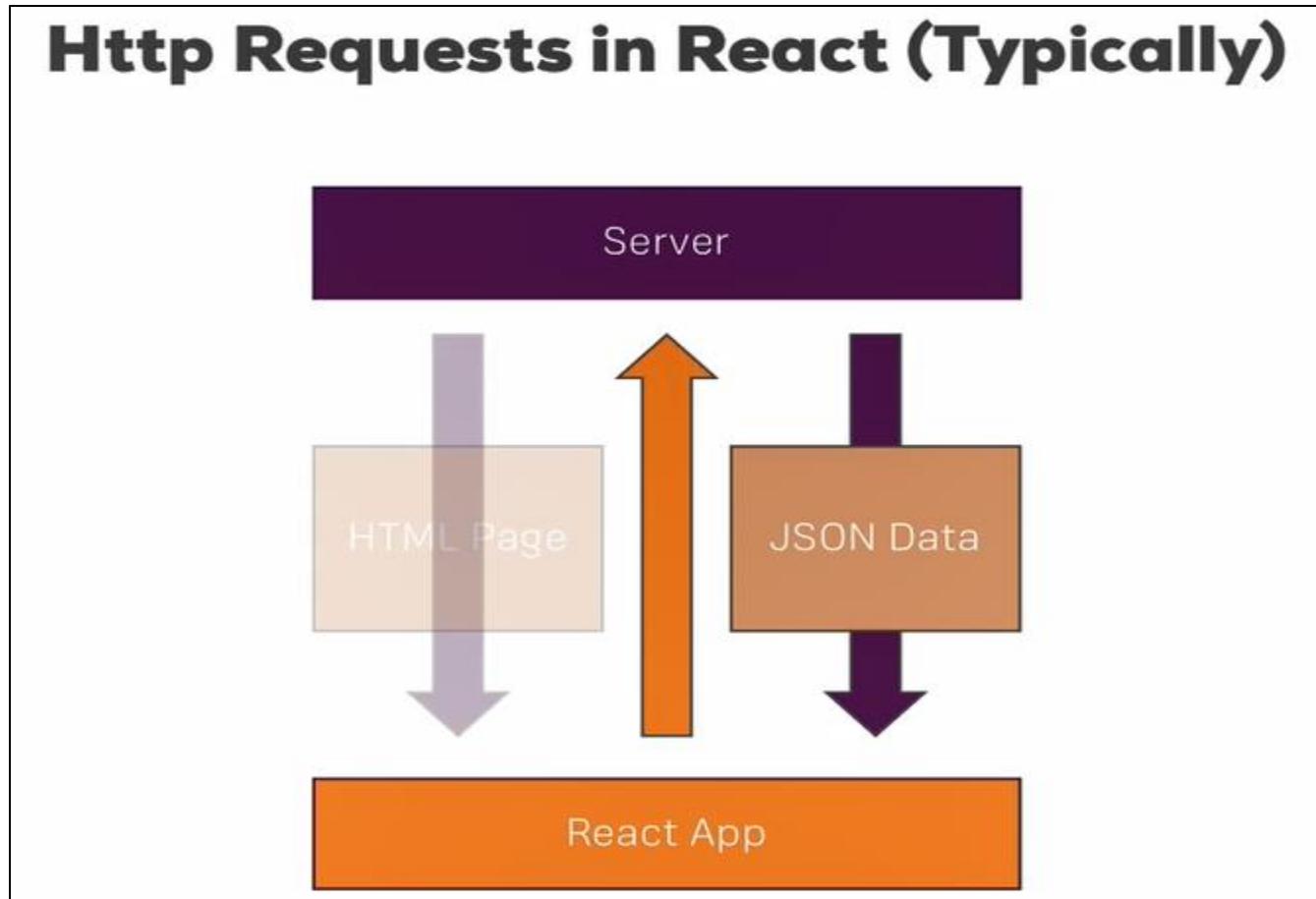
9. Reaching out to the Web (Http Ajax)

9.1. Module Introduction

9.2. Understanding Http Requests in React

In response to request sent to server, react gets json data (typical single page applications).

In response to request sent to server, react gets html data (typical multi page applications).



9.3. Understanding our Project and Introducing Axios

Jsonplaceholder as backend rest service.

Axios can be used in any javascript code.

>> *npm install axios –save*

9.4. Creating a Http Request to GET Data

`componentDidMount() → best place to send HTTP request (side effect)`

9.5. Rendering Fetched Data to the Screen

Display post data on the screen.

Blog.js

```
import React, { Component } from "react";
import Post from "../../components/Post/Post";
import FullPost from "../../components/FullPost/FullPost";
import NewPost from "../../components/NewPost/NewPost";
import "./Blog.css";
import axios from "../../../../../node_modules/axios";

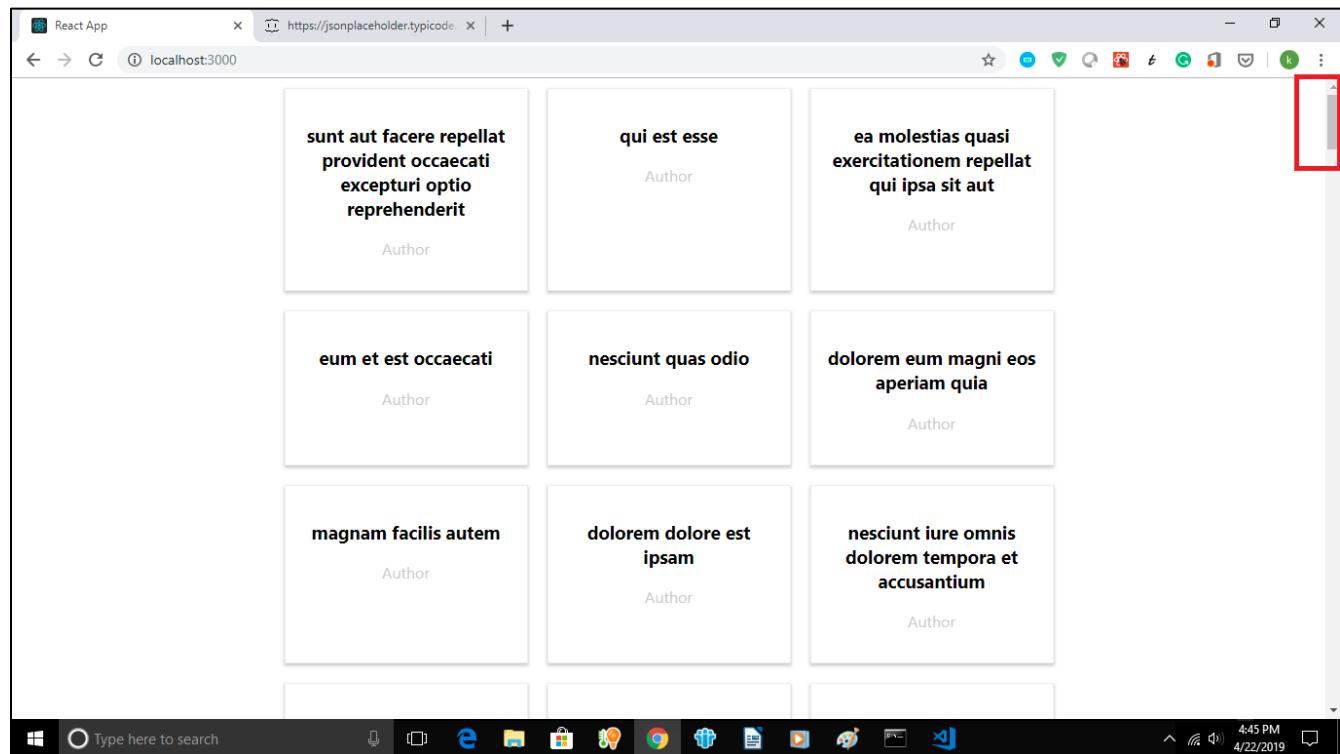
class Blog extends Component {
    // create state to dynamically update the DOM
    state = {
        posts: []
    };
    componentDidMount() {
        axios.get("https://jsonplaceholder.typicode.com/posts").then(response => {
            this.setState({ posts: response.data });
            // console.log(response);
        });
    }
    render() {
        // create map and return JSX code
        const posts = this.state.posts.map(post => {
            return <Post key={post.id} title={post.title} />;
        });
        return (
            <div>
                <section className="Posts">{posts}</section>
                <section>
                    <FullPost />
                </section>
                <section>
                    <NewPost />
                </section>
            </div>
        );
    }
}
```

```
export default Blog;
```

Post.js

```
import React from "react";
import "./Post.css";

const post = props => (
  <article className="Post">
    <h1>{props.title}</h1>
    <div className="Info">
      <div className="Author">Author</div>
    </div>
  </article>
);
export default post;
```



9.6. Transforming Data

Transform the response data we get from axios request.

Blog.js

```

import React, { Component } from "react";
import Post from "../../components/Post/Post";
import FullPost from "../../components/FullPost/FullPost";
import NewPost from "../../components/NewPost/NewPost";
import "./Blog.css";
import axios from "../../../../../node_modules/axios";

class Blog extends Component {
    // create state to dynamically update the DOM
    state = {
        posts: []
    };
    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: "Kunj"
                };
            });
            this.setState({ posts: updatedPosts });
        });
    }
    render() {
        // create map and return JSX code
        const posts = this.state.posts.map(post => {
            return <Post key={post.id} title={post.title}
author={post.author} />;
        });
        return (
            <div>
                <section className="Posts">{posts}</section>
                <section>
                    <FullPost />
                </section>
                <section>
                    <NewPost />
                </section>
            </div>
        );
    }
}

```

```

        }
    }
export default Blog;

```

Post.js

```

import React from "react";
import "./Post.css";
const post = props => (
    <article className="Post">
        <h1>{props.title}</h1>
        <div className="Info">
            <div className="Author">{props.author}</div>
        </div>
    </article>
);
export default post;

```

O/P:

The screenshot shows a web application running on localhost:3000. At the top, there are four cards representing blog posts. Each card contains placeholder Latin text and the name 'Kunj' as the author. Below the cards is a form titled 'Add a Post' with fields for 'Title', 'Content', and 'Author'. The 'Author' field is populated with 'Max'. An 'Add Post' button is at the bottom of the form.

9.7. Making a Post Selectable

Click on one of the posts and load the post data in the div below the posts.

- 1) Add 'onClick' event to article element in Post.js

Post.js

```
import React from "react";
import "./Post.css";
const post = props => (
  <article className="Post" onClick={props.clicked}>
    <h1>{props.title}</h1>
    <div className="Info">
      <div className="Author">{props.author}</div>
    </div>
  </article>
);
export default post;
```

2) Pass 'clicked' property in Blog.js <Post> element and write handler method

Blog.js

```
import React, { Component } from "react";
import Post from "../../components/Post/Post";
import FullPost from "../../components/FullPost/FullPost";
import NewPost from "../../components/NewPost/NewPost";
import "./Blog.css";
import axios from "../../../../../node_modules/axios";

class Blog extends Component {
  // create state to dynamically update the DOM
  state = {
    posts: [],
    selectedPostId: null
  };
  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: "Kunj"
        };
      });
      this.setState({ posts: updatedPosts });
    });
  }
}
```

```

        });
    }
    postSelectedHandler = id => {
        this.setState({ selectedPostId: id });
    };
    render() {
        // create map and return JSX code
        const posts = this.state.posts.map(post => {
            return (
                <Post
                    key={post.id}
                    title={post.title}
                    author={post.author}
                    clicked={() => this.postSelectedHandler(post.id)}
                />
            );
        });
        return (
            <div>
                <section className="Posts">{posts}</section>
                <section>
                    <FullPost id={this.state.selectedPostId} />
                </section>
                <section>
                    <NewPost />
                </section>
            </div>
        );
    }
}
export default Blog;

```

3) Add condition in FullPost.js

```

import React, { Component } from "react";
import "./FullPost.css";
class FullPost extends Component {
    render() {
        let post = <p style={{ textAlign: "center" }}>Please select a
Post!</p>;

```

```

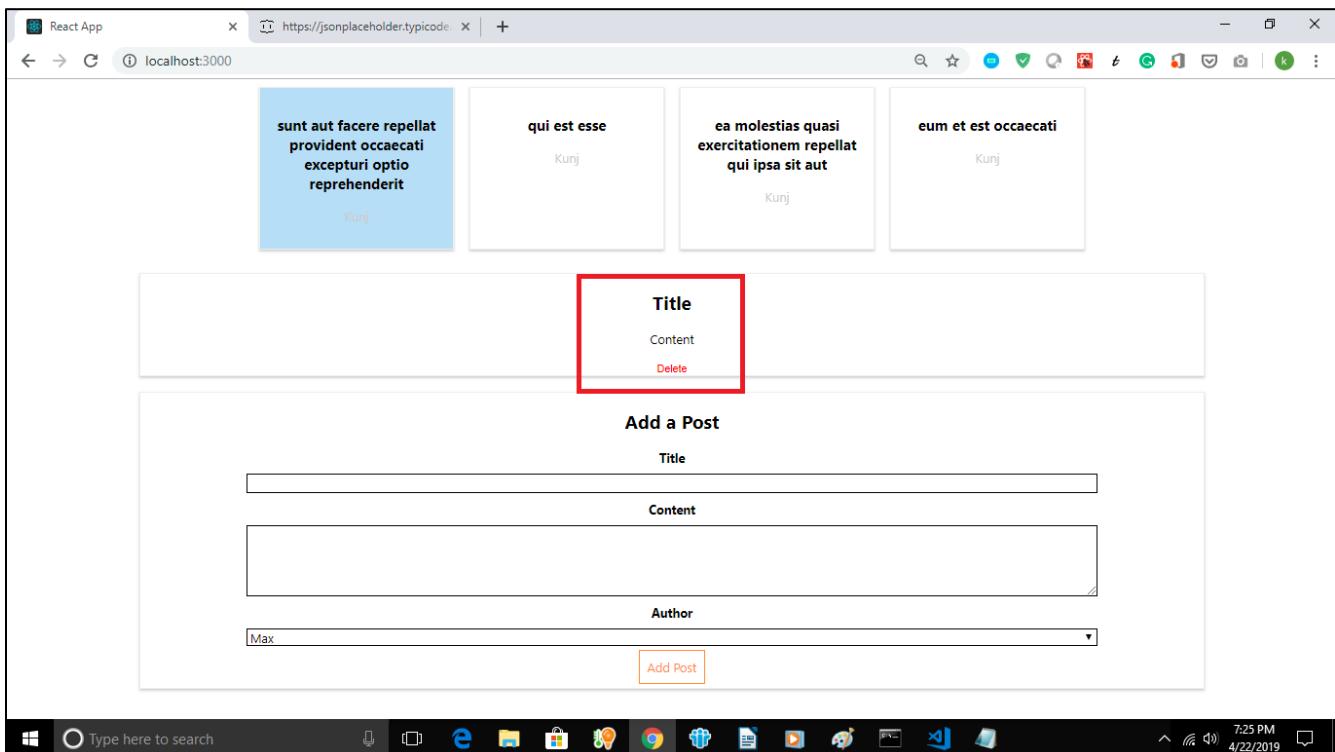
        // if the id is null, show the above message, otherwise show the
below post
    if (this.props.id) {
        post = (
            <div className="FullPost">
                <h1>Title</h1>
                <p>Content</p>
                <div className="Edit">
                    <button className="Delete">Delete</button>
                </div>
            </div>
        );
    }
    return post;
}
export default FullPost;

```

Default O/P:

The screenshot shows a web browser window with the URL `https://jsonplaceholder.typicode.com` in the address bar. The page displays four cards representing posts, each with a title, content, and author information. Below the cards is a red-bordered box containing the text "Please select a Post!". Underneath this box is a form titled "Add a Post" with fields for "Title", "Content", and "Author". The "Author" field has the value "Max" and is highlighted with a red border. At the bottom right of the form is a button labeled "Add Post". The browser's taskbar at the bottom shows various pinned icons.

O/P after clicking one of the 4 posts:



Note: null is treated as false.

In the next lecture, also fetch the data of the selected post into the div below the posts.

9.8. Fetching Data on Update (without Creating Infinite Loops)

Q) Which lifecycle hook should be used here?

If we use componentDidUpdate() to update state, we update the component again and enter an infinite loop. So we need to take care of this infinite loop.

FullPost.js

```
import React, { Component } from "react";
import "./FullPost.css";
import Axios from "axios";

class FullPost extends Component {
  state = {
    loadedPost: null
  };

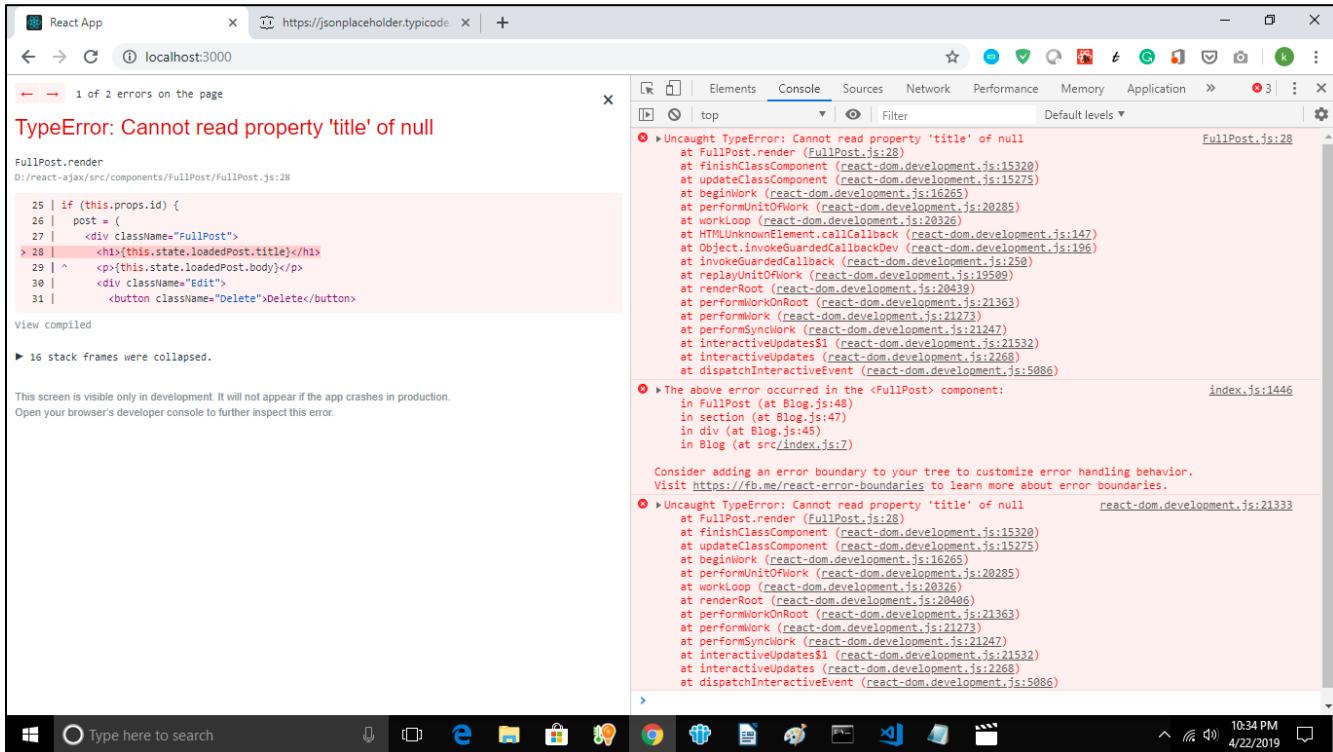
  componentDidUpdate() {
    //if id is not null
  }
}
```

```

if (this.props.id) {
  Axios.get(
    "https://jsonplaceholder.typicode.com/posts/" +
  this.props.id
    ).then(response => {
      this.setState({ loadedPost: response.data });
    });
}
render() {
  let post = <p style={{ textAlign: "center" }}>Please select a
Post!</p>;
  // if the id is null, show the above message, otherwise show the
below post
  if (this.props.id) {
    post = (
      <div className="FullPost">
        <h1>{this.state.loadedPost.title}</h1>
        <p>{this.state.loadedPost.body}</p>
        <div className="Edit">
          <button className="Delete">Delete</button>
        </div>
      </div>
    );
  }
  return post;
}
export default FullPost;

```

O/P:



Q) Why are we getting error in the above screenshot?

We are getting a valid id property first(`if(this.props.id)` in `componentDidUpdate()`) and in the `render()` method even before we actually have a valid post (`this.setState({ loadedPost: response.data });`). Javascript executes code asynchronously.

How to fix this?

FullPost.js

```
import React, { Component } from "react";
import "./FullPost.css";
import Axios from "axios";

class FullPost extends Component {
  state = {
    loadedPost: null
  };

  componentDidUpdate() {
    //if id is not null
    if (this.props.id) {
      Axios.get(`https://jsonplaceholder.typicode.com/posts/${this.props.id}`)
        .then(response => {
          this.setState({ loadedPost: response.data });
        })
        .catch(error => {
          console.log(error);
        });
    }
  }
}
```

```

        "https://jsonplaceholder.typicode.com/posts/" +
this.props.id
    ).then(response => {
      this.setState({ loadedPost: response.data });
    });
  }
}

render() {
  let post = <p style={{ textAlign: "center" }}>Please select a
Post!</p>;

  if (this.props.id) {
    post = <p style={{ textAlign: "center" }}>Loading...</p>;
  }
  // As soon as we have a post, this if block will execute
  if (this.state.loadedPost) {
    post = (
      <div className="FullPost">
        <h1>{this.state.loadedPost.title}</h1>
        <p>{this.state.loadedPost.body}</p>
        <div className="Edit">
          <button className="Delete">Delete</button>
        </div>
      </div>
    );
  }
  return post;
}
export default FullPost;

```

Note: There is a problem in the above code despite the fact that we are able to see the clicked post in the div. The request is going infinitely to the rest endpoint.

Why?

Bcz we are updating the state from within the componentDidUpdate(), hence updating the component.

How to fix this?

FullPost.js

```
componentDidUpdate() {
    //if id is not null
    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 => {
            this.setState({ loadedPost: response.data });
        });
    }
}
```

9.9. POSTing Data to the Server

NewPost.js

```
import React, { Component } from "react";
import "./NewPost.css";
import axios from "axios";

class NewPost extends Component {
    state = {
        title: "",
        content: "",
        author: "Max"
    };

    postDataHandler = () => {
        const data = {
            title: this.state.title,
            body: this.state.content,
            author: this.state.author
        };
        axios
            .post("https://jsonplaceholder.typicode.com/posts", data)
```

```
.then(response => {
  console.log(response);
});
};

render() {
  return (
    <div className="NewPost">
      <h1>Add a Post</h1>
      <label>Title</label>
      <input
        type="text"
        value={this.state.title}
        onChange={event => this.setState({ title:
event.target.value })}
      />
      <label>Content</label>
      <textarea
        rows="4"
        value={this.state.content}
        onChange={event => this.setState({ content:
event.target.value })}
      />
      <label>Author</label>
      <select
        value={this.state.author}
        onChange={event => this.setState({ author:
event.target.value })}
      >
        <option value="Max">Max</option>
        <option value="Manu">Manu</option>
      </select>
      <button onClick={this.postDataHandler}>Add Post</button>
    </div>
  );
}

export default NewPost;
```

9.10. Sending a DELETE Request

FullPost.js

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

import "./FullPost.css";
import Axios from "axios";

class FullPost extends Component {
  state = {
    loadedPost: null
  };

  componentDidUpdate() {
    //if id is not null
    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 => {
        this.setState({ loadedPost: response.data });
      });
    }
  }

  deletePostHandler = () => {
    Axios.delete(
      "https://jsonplaceholder.typicode.com/posts/" + this.props.id
    ).then(response => {
      console.log(response);
    });
  };

  render() {
    let post = <p style={{ textAlign: "center" }}>Please select a
Post!</p>;
  }
}
```

```

        if (this.props.id) {
            post = <p style={{ textAlign: "center" }}>Loading...</p>;
        }
        // As soon as we have a post, this if block will execute
        if (this.state.loadedPost) {
            post = (
                <div className="FullPost">
                    <h1>{this.state.loadedPost.title}</h1>
                    <p>{this.state.loadedPost.body}</p>
                    <div className="Edit">
                        <button onClick={this.deletePostHandler}>
                            Delete
                        </button>
                    </div>
                </div>
            );
        }
        return post;
    }
}

export default FullPost;

```

9.11. Fixing a Bug

9.12. Handling Errors Locally

- a) Send an incorrect URL in blog.js
- b) write catch in axios request
- c) Add an error property to the state object

Blog.js

```

import React, { Component } from "react";

import Post from "../../components/Post/Post";
import FullPost from "../../components/FullPost/FullPost";
import NewPost from "../../components/NewPost/NewPost";

```

```

import "./Blog.css";
import axios from "../../node_modules/axios";

class Blog extends Component {
    // create state to dynamically update the DOM
    state = {
        posts: [],
        selectedPostId: null,
        error: false
    };
    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: "Kunj"
                    };
                });
                this.setState({ posts: updatedPosts });
            })
            .catch(error => {
                this.setState({ error: true });
            });
    }

    postSelectedHandler = id => {
        this.setState({ selectedPostId: id });
    };

    render() {
        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}
                )
            });
        }
        return (
            <div>
                {posts}
            </div>
        );
    }
}

```

```

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

return (
  <div>
    <section className="Posts">{posts}</section>
    <section>
      <FullPost id={this.state.selectedPostId} />
    </section>
    <section>
      <NewPost />
    </section>
  </div>
);
}

export default Blog;

```

9.13. Adding Interceptors to Execute Code Globally

Handling errors locally in components make lot of sense bcz you want to do different things with your errors depending on which component you are in.

But sometimes, at some places you would want to execute some code globally. No matter which Http request you send within which component, you want to do something when that request gets sent or when the response returns. You can do this with axios with the help of **Interceptors**.

Interceptors are functions you can define globally that will be executed for every request leaving your app and every response returning into it. This is useful in:

1) Setting up some header (authorization)

2) Log responses

3) Handling errors globally

Go to index.js (here we start our react app by mounting it to DOM) and do the followings:

1) import axios → All axios imports withing the application share the same configuration

2) define request and response interceptors

Index.js

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import Blog from "./containers/Blog/Blog";
import * as serviceWorker from "./serviceWorker";
import axios from "../node_modules/axios";

// this interceptors object will be shared across all files in the project
// it will intercept all the http request in the application
axios.interceptors.request.use(
  request => {
    console.log(request);
    // You can also Edit request config here
    /*
      You need to always return the request config from the
      interceptor, otherwise you are blocking the request
    */
    return request;
  },
  error => {
    // error function will handle all the errors
    console.log(error);
    /*
      We need to return this error so that we forward it to the request
      made in the component and handle in the catch method
    */
    return Promise.reject(error);
  }
);

// To handle responses
axios.interceptors.response.use(
  response => {
    console.log(response);
    // Edit response config
    return response;
  }
);
```

```

    },
    error => {
      console.log(error);
      return Promise.reject(error);
    }
);

ReactDOM.render(<Blog />, document.getElementById("root"));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();

```

9.14. Removing Interceptors

You learned how to add an interceptor, getting rid of one is also easy. Simply store the reference to 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>):

```

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

```

9.15. Setting a Default Global Configuration for Axios

index.js

```

// global config for all the requests
axios.defaults.baseURL = "https://jsonplaceholder.typicode.com";
axios.defaults.headers.common["Authorization"] = "AUTH TOKEN";
axios.defaults.headers.post["Content-Type"] = "application/json";

```

Axios.get("https://jsonplaceholder.typicode.com/posts/") will be converted to Axios.get("/posts/").

9.16. Creating and Using Axios Instances

What if you do not want to use the prev configuration for all the url. What if you want some urls to be having different base url, different headers.

In such cases you can use instances.

Create a file `axios.js` in `src` folder.

```
import axios from "../../node_modules/axios";

// creates an instance of axios object
// you can create multiple such instances in multiple files
// Pass javascript object to configure it
const instance = axios.create({
  baseURL: "https://jsonplaceholder.typicode.com"
});

/* this will override default configuration
in index.js for all the request sent using this instance
*/
instance.defaults.headers.common["Authorization"] = "AUTH TOKEN FROM
INSTANCE";

// you can also add interceptors to this instance (just like default axios
object)
// instance.interceptors.request.....
// To use it in other files
export default instance;
```

This axios.js file can be imported in other files as the below:

```
import axios from ".././axios";
```

9.17. Wrap Up

9.18. Useful Resources _ Links

11. Multi-Page-Feeling in a Single-Page-App Routing

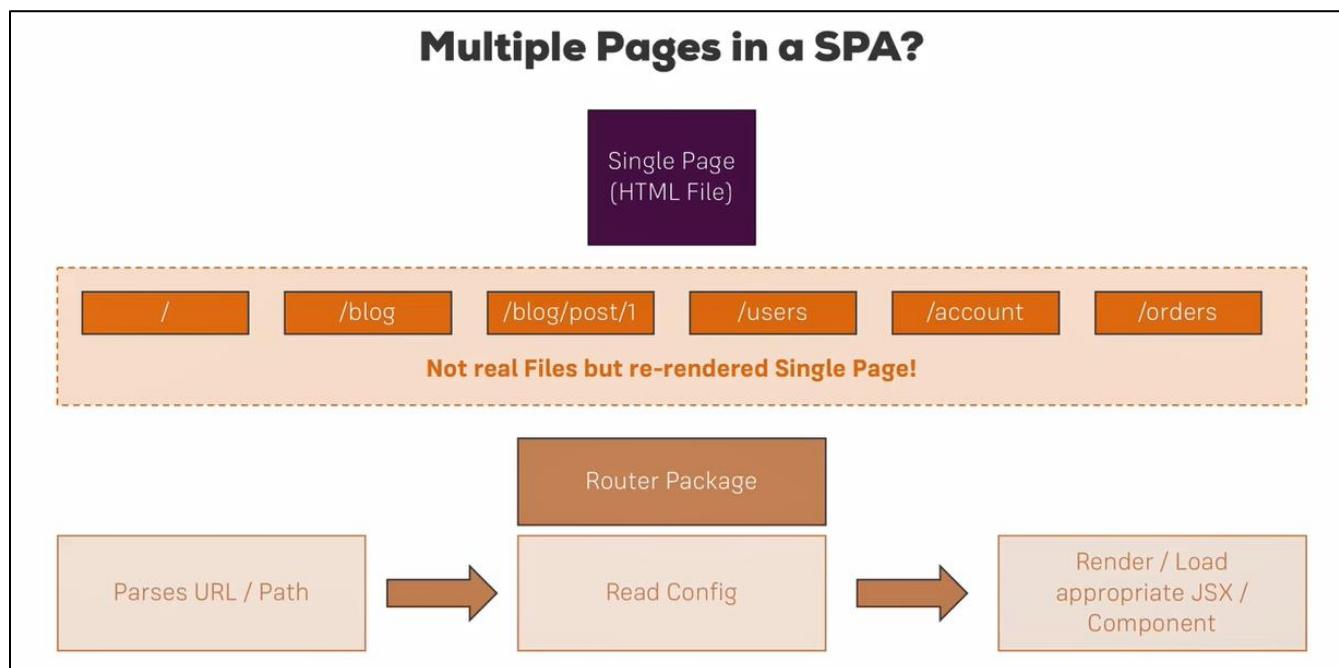
11.1. Module Introduction

11.2. Routing and SPAs

To give multipage experience to users, we do not create multiple html files, instead we use javascript to render different parts (or the entire) of the single html page.

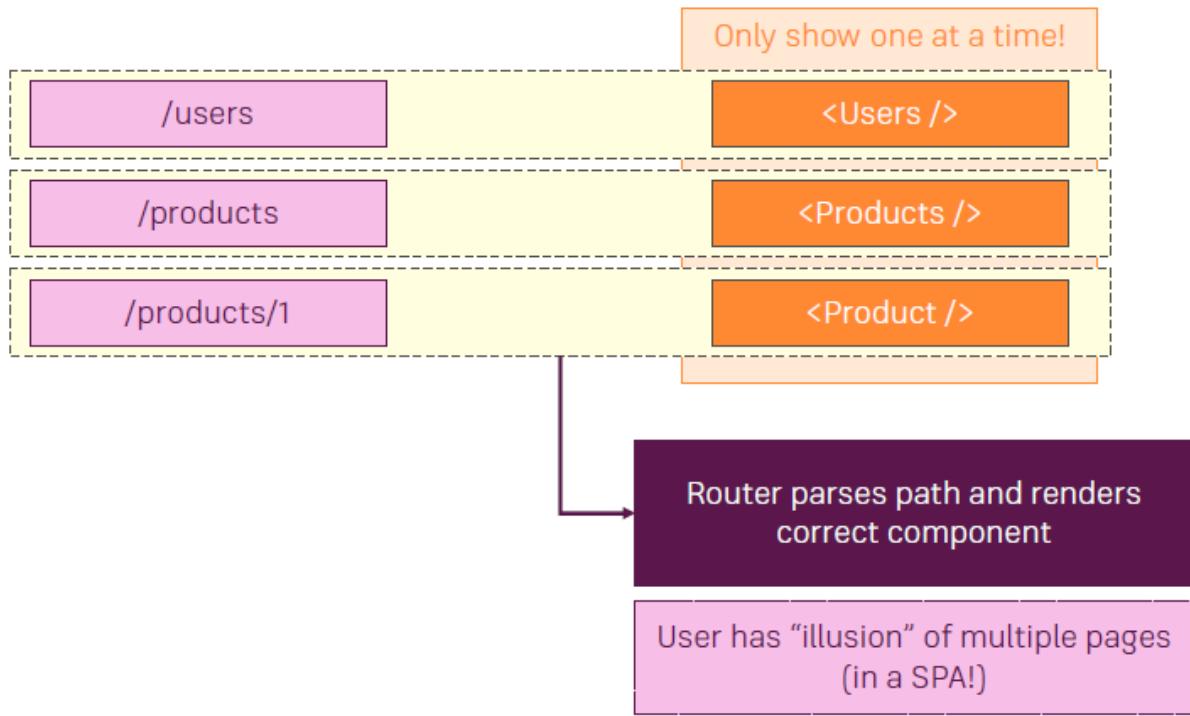
Use Router Package to achieve this. Router package does the following:

- 1) Parses URL/Path to understand where the user want to go to. We developer have to configure different paths in our application
- 2) Reads our configuration to know what should happen when users visits one of these paths
- 3) Render appropriate JSX/Component code depending on which path the user visited



11.2.1 routing-learning-card.pdf

Routing



11.3. Setting Up Links

Allow users to click on some link and load different pages. (Use Blog project)

create links in blog.js and create styles for them.

Blog.js

```
import React, { Component } from "react";
import axios from "../../axios";

import Post from "../../components/Post/Post";
import FullPost from "../../components/FullPost/FullPost";
import NewPost from "../../components/NewPost/NewPost";
import "./Blog.css";

class Blog extends Component {
  state = {
    posts: [],
    selectedPostId: null,
    error: false
  }
}
```

```

};

componentDidMount() {
    axios
        .get("/posts")
        .then(response => {
            const posts = response.data.slice(0, 4);
            const updatedPosts = posts.map(post => {
                return {
                    ...post,
                    author: "Max"
                };
            });
            this.setState({ posts: updatedPosts });
            // console.log( response );
        })
        .catch(error => {
            // console.log(error);
            this.setState({ error: true });
        });
}

postSelectedHandler = id => {
    this.setState({ selectedPostId: id });
};

render() {
    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}
                    author={post.author}
                    clicked={() => this.postSelectedHandler(post.id)}
                />
            );
        });
    }
}

```

```

        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>
                <section className="Posts">{posts}</section>
                <section>
                    <FullPost id={this.state.selectedPostId} />
                </section>
                <section>
                    <NewPost />
                </section>
            </div>
        );
    }
}

export default Blog;

```

Blog.css

```

.Posts {
    display: flex;
    flex-flow: row wrap;
    justify-content: center;
    width: 80%;
    margin: auto;
}

.Blog ul {
    list-style: none;
}

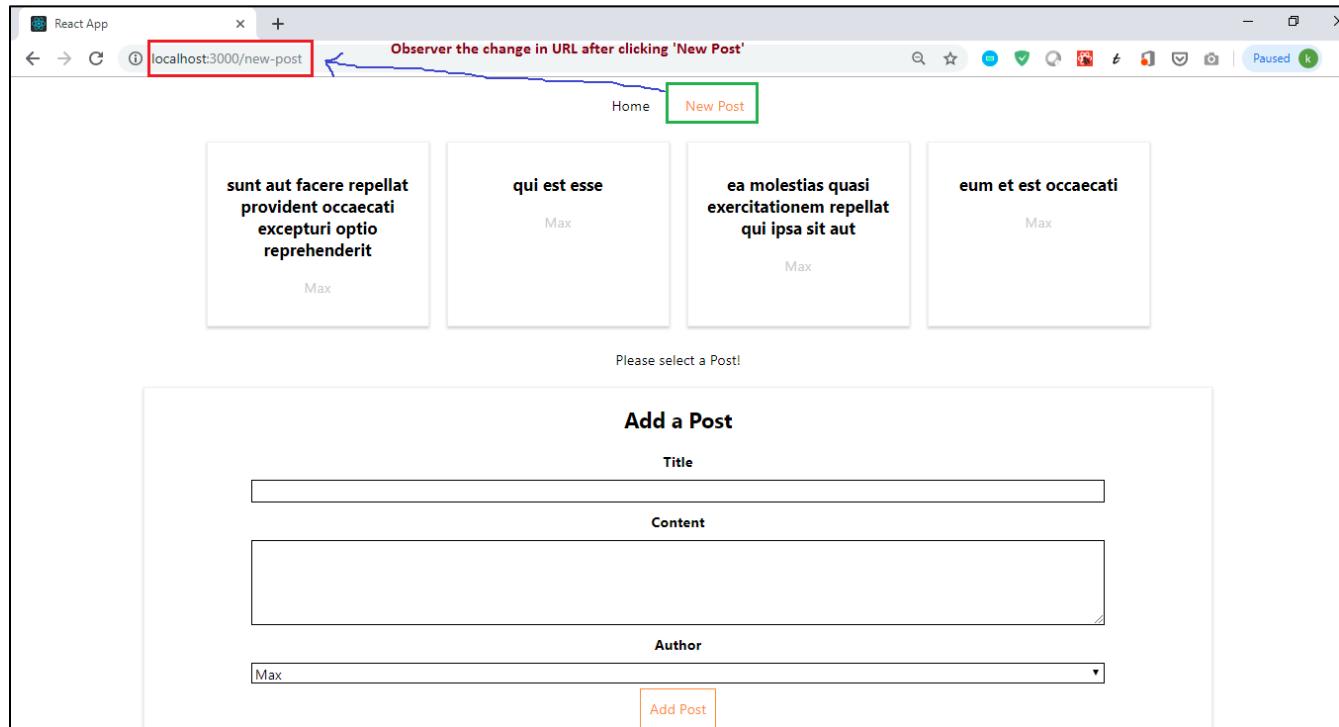
```

```
margin: 0;
padding: 0;
width: 100%;
text-align: center;
}

.Blog li {
  display: inline-block;
  margin: 20px;
}
.Blog a {
  text-decoration: none;
  color: black;
}

.Blog a:hover,
.Blog a:active {
  color: #fa923f;
}
```

O/P:



4. Setting Up the Router Package

D:\react-routing-project> `npm install --save react-router react-router-dom`

These packages are not created by facebook. These are de facto for routing in react application.

For root route (/), only load the posts.

1) Remove Fullpost and Newpost from Blog component

2) Only load posts when we are at the root url (/)

3) To achieve (2), first enable routing in react app. This can be done either in index.js or App.js. There you need to wrap the part of your app which should be able to render routes and to read routes. You need to wrap it with a component you import from react-router-dom package.

Move FullPost and newPost into Blog folder. Create a new folder 'Posts' in Blog' and create a Posts.js in Posts folder.

App.js

```
import React, { Component } from "react";
import { BrowserRouter } from "react-router-dom";
import Blog from "./containers/Blog/Blog";

class App extends Component {
  render() {
    return (
      <BrowserRouter>
        <div className="App">
          <Blog />
        </div>
      </BrowserRouter>
    );
  }
}
export default App;
```

5. 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.

6. Preparing the Project For Routing

Remove `<section className="Posts">{posts}</section>` from Blog.js and put in a new component, Posts.js.

Move state from Blog.js to Posts.js. Move posts logic (in render() of Blog.js) to Posts.js render(). Also move postSelectedHandler(), componentDidMount().

Posts.js

```
import React, { Component } from "react";
import axios from "../../axios";
import Post from "../../components/Post/Post";
import "./Posts.css";

class Posts extends Component {
  state = {
    posts: []
  };

  componentDidMount() {
    axios
      .get("/posts")
      .then(response => {
        const posts = response.data.slice(0, 4);
        const updatedPosts = posts.map(post => {
          return {
            ...post,
            author: "Max"
          };
        });
        this.setState({ posts: updatedPosts });
        // console.log( response );
      })
      .catch(error => {
        console.log(error);
        // this.setState({ error: true });
      });
  }
}
```

```

}

postSelectedHandler = id => {
    this.setState({ selectedPostId: id });
};

render() {
    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}
                    author={post.author}
                    clicked={() => this.postSelectedHandler(post.id)}
                />
            );
        });
    }
    return <section className="Posts">{posts}</section>;
}
}

export default Posts;

```

Blog.js

```

import React, { Component } from "react";
import axios from "../../axios";
import "./Blog.css";
import Posts from "./Posts/Posts";

class Blog extends Component {
    componentDidMount() {
        axios
            .get("/posts")
            .then(response => {
                const posts = response.data.slice(0, 4);

```

```

        const updatedPosts = posts.map(post => {
            return {
                ...post,
                author: "Max"
            };
        });
        this.setState({ posts: updatedPosts });
        // console.log( response );
    })
    .catch(error => {
        // console.log(error);
        this.setState({ error: true });
    });
}

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>
    );
}
}

export default Blog;

```

Posts.css

```
.Posts {  
  display: flex;  
  flex-flow: row wrap;  
  justify-content: center;  
  width: 80%;  
  margin: auto;  
}
```

Blog.css

```
.Blog ul {  
  list-style: none;  
  margin: 0;  
  padding: 0;  
  width: 100%;  
  text-align: center;  
}  
  
.Blog li {  
  display: inline-block;  
  margin: 20px;  
}  
  
.Blog a {  
  text-decoration: none;  
  color: black;  
}  
  
.Blog a:hover,  
.Blog a:active {  
  color: #fa923f;  
}
```

o/p: (only Posts are visible)



7. Setting Up and Rendering Routes

Load Posts dynamically depending on the path we have in our URL.

In Blog.js,

(1) `import { Route } from "react-router-dom";`

(2) Add below in render() of blog.js

```
<Route path="/" render={() => <h1>Home</h1>} />
```

O/P:



No matter whether you click on ‘Home’ or ‘New Post’, you always see the ‘Home’ in bold. The reason is that `path="/"` just checks if the path starts with “/”. To change this behavior add keyword ‘exact’ in `<Route>`.

The below code will render:

```
<Route path="/" exact render={() => <h1>Home</h1>} />
```

```
<Route path="/" render={() => <h1>Home 2</h1>} />
```

O/P:



8. Rendering Components for Routes

```
{/* this will load Posts component */}
```

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

O/P:



9. Switching Between Pages

Blog.js

```
import React, { Component } from "react";
import axios from "../../axios";
import "./Blog.css";
import Posts from "./Posts/Posts";
import { Route } from "react-router-dom";
import NewPost from "./NewPost/NewPost";
```

```

class Blog extends Component {
  componentDidMount() {
    axios
      .get("/posts")
      .then(response => {
        const posts = response.data.slice(0, 4);
        const updatedPosts = posts.map(post => {
          return {
            ...post,
            author: "Max"
          };
        });
        this.setState({ posts: updatedPosts });
        // console.log( response );
      })
      .catch(error => {
        // console.log(error);
        this.setState({ error: true });
      });
  }

  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>
        {/* <Route path="/" exact render={() =>
<h1>Home</h1>} />

```

```

<Route path="/" render={() => <h1>Home 2</h1>} /> */
      /* this will load Posts component */
      <Route path="/" exact component={Posts} />
      <Route path="/new-post" component={NewPost} />
    </div>
  );
}
export default Blog;

```

Note: The issue with this approach is that everytime we click on link, page reloads. A reloading application means that javascript code starting anew therefore all previous application state is lost. To be fixed in the next lecture.

10. Using Links to Switch Pages

Blog.js

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

2) Replace <a> with <Link>

With the above changes react will re-render the pages, it will not reload them.

Blog.js

```

import React, { Component } from "react";
import axios from "../../axios";
import "./Blog.css";
import Posts from "./Posts/Posts";
import { Route, Link } from "react-router-dom";
import NewPost from "./NewPost/NewPost";

class Blog extends Component {
  componentDidMount() {
    axios
      .get("/posts")
      .then(response => {

```

```

        const posts = response.data.slice(0, 4);
        const updatedPosts = posts.map(post => {
            return {
                ...post,
                author: "Max"
            };
        });
        this.setState({ posts: updatedPosts });
        // console.log( response );
    })
    .catch(error => {
        // console.log(error);
        this.setState({ error: true });
    });
}

render() {
    return (
        <div className="Blog">
            <header>
                <nav>
                    <ul>
                        <li>
                            <Link to="/">Home</Link>
                        </li>
                        <li>
                            <Link
                                to={{
                                    pathname: "/new-post",
                                    hash: "#submit", // To jump
to any point
                                search: "?quick-submit=true"
                            }}>
                            New Post
                        </li>
                    </ul>
                </nav>
            </header>
        </div>
    );
}

```

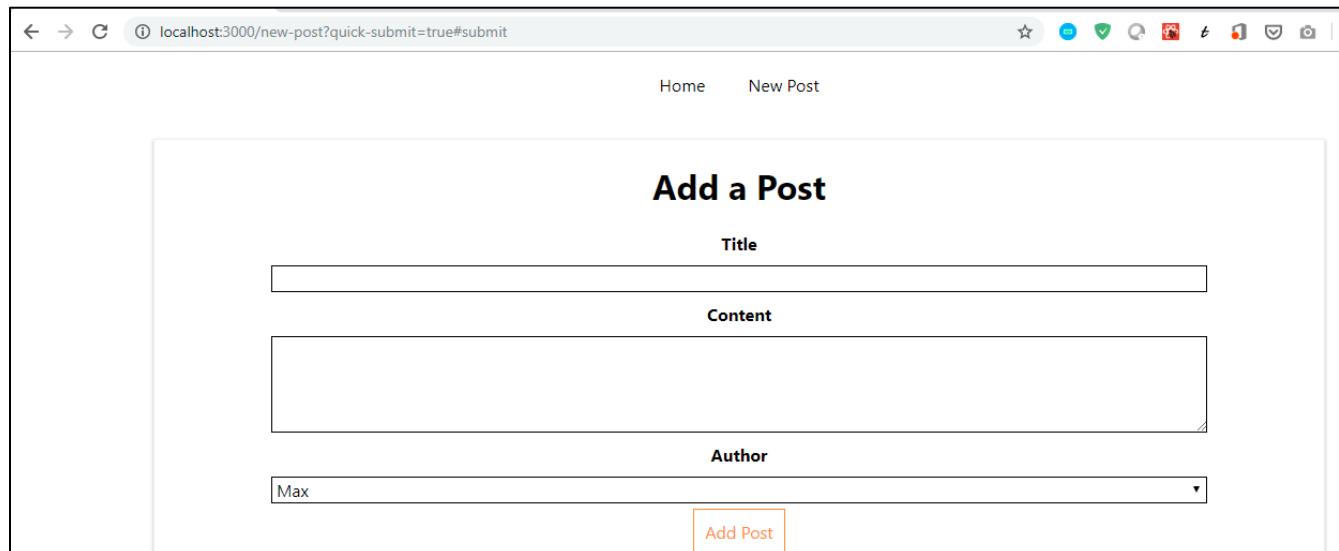
```

        </nav>
    </header>
    {/* <Route path="/" exact render={() =>
<h1>Home</h1>} />
<Route path="/" render={() => <h1>Home 2</h1>} /> */}
        {/* this will load Posts component */}
        <Route path="/" exact component={Posts} />
        <Route path="/new-post" component={NewPost} />
    </div>
);
}
}

export default Blog;

```

O/P:



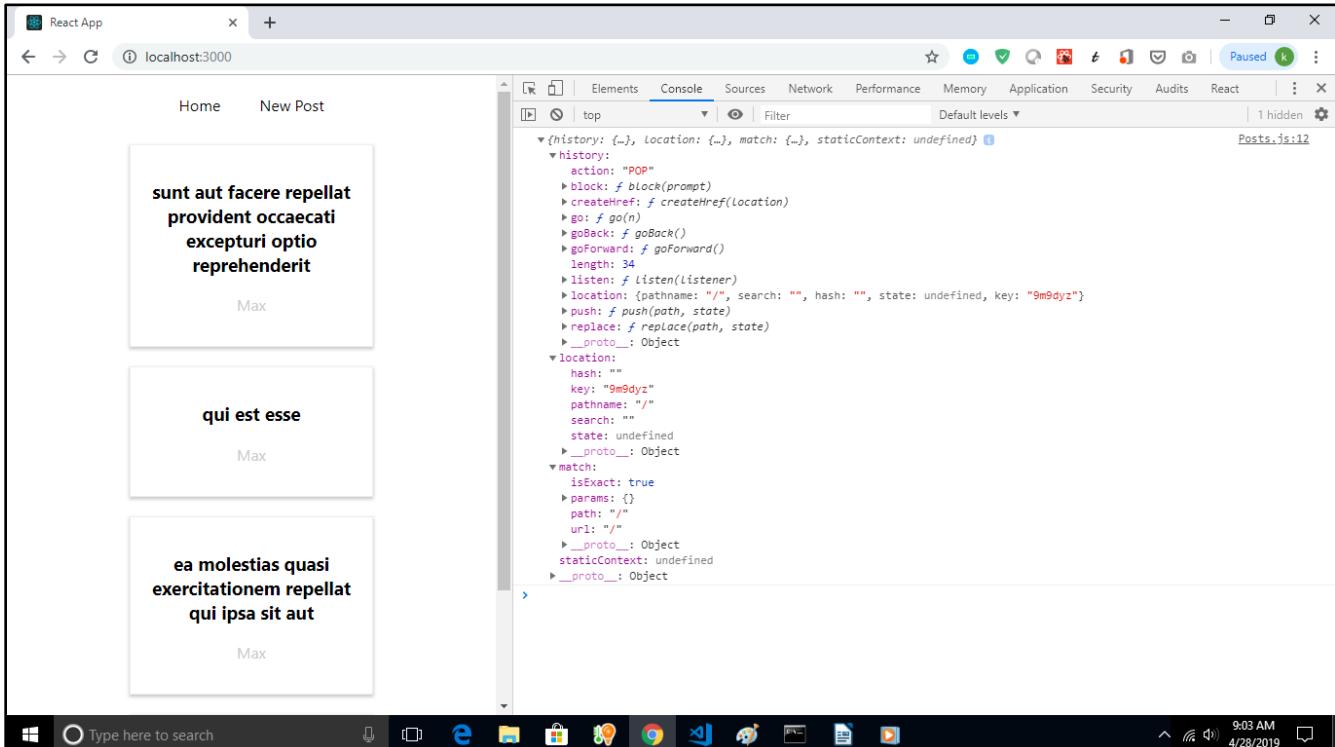
11. Using Routing-Related Props

react-router gives extra information about the loaded route through 'props'.

In Posts.js, in componentDidMount(), add below to see the info:

```
console.log(props);
```

O/P:



12. The withRouter HOC _ Route Props

What if we want to get this info (react-router gives extra information) not in one of containers, components loaded through `<Route>` (as defined in `Blog.js`) but in a component loaded as a part of such a container like `Posts` container.

Turn Post component in one that returns JSX code. Add `console.log(posts)`. Observe in the screenshot that routing related components are not passed down the component tree.

We cannot access them in components which we just embed as part of the JSX code of a container.

Post.js

```
import React from "react";
import "./Post.css";

const post = props => {
  console.log(props);
  return (
    <article className="Post" onClick={props.clicked}>
      <h1>{props.title}</h1>
```

```

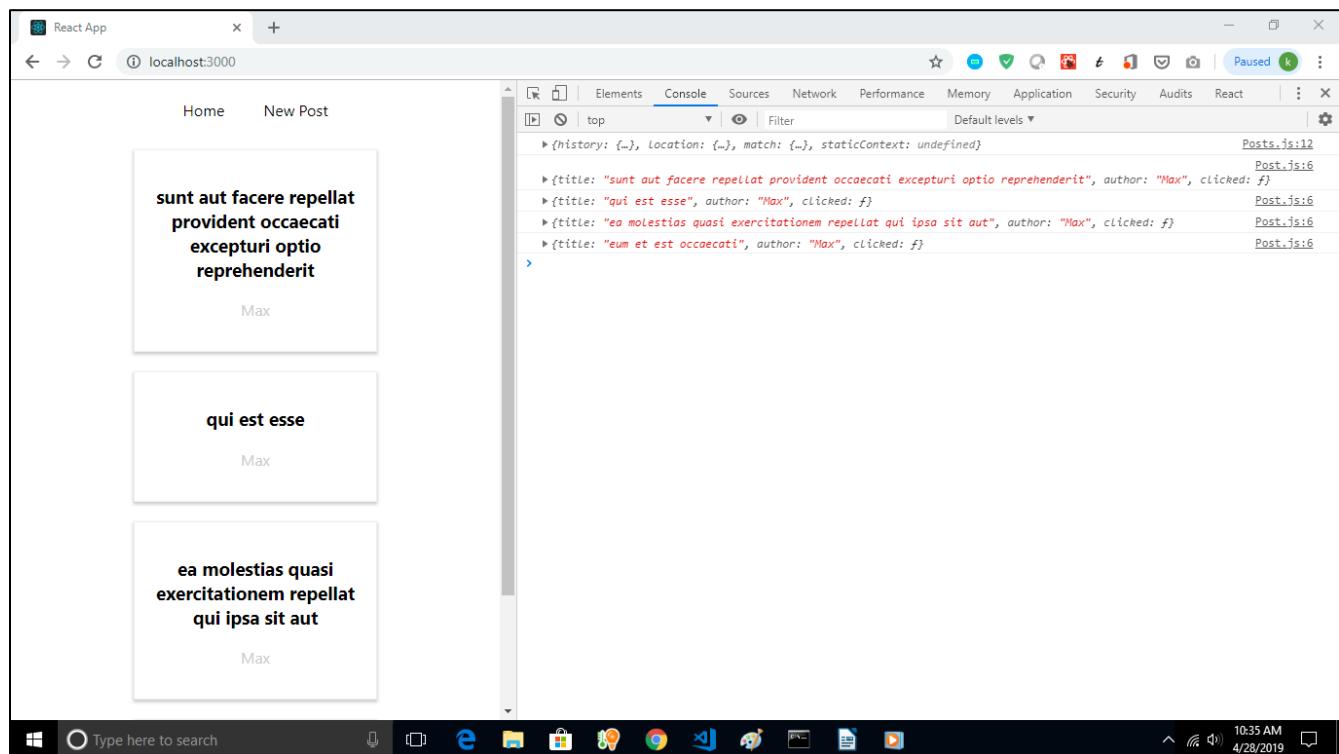
        <div className="Info">
            <div className="Author">{props.author}</div>
        </div>
    </article>
);

};

export default post;

```

O/P:



If you want to see react-router properties in the above code, there are 2 ways:

1) Pass them on from Posts.js (...this.props)

Alternatively, pass match={this.props.match}.

Posts.js

```

render() {
    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}
        author={post.author}
        // To access react-router extra info in Post.js
        {...this.props}
        clicked={() => this.postSelectedHandler(post.id)}
      />
    );
  });
}

return <section className="Posts">{posts}</section>;
}

```

2) Use Higher order component (HOC) on the Post component. withRouter adds

Post.js

a) `import { withRouter } from "react-router-dom";`

b) withRouter is a way to make a component route aware and it will get the props containing the info for the nearest loaded route. In this case, since Post is included in Posts, you get the same props as we receive in Posts.

Post.js

```

import React from "react";
import { withRouter } from "react-router-dom";
import "./Post.css";

const post = props => {
  console.log(props);
  return (
    <article className="Post" onClick={props.clicked}>
      <h1>{props.title}</h1>
      <div className="Info">
        <div className="Author">{props.author}</div>
      </div>
    </article>
  );
}

export default withRouter(Post);

```

```
        </article>
    );
};

export default withRouter(post);
```

13. Absolute vs Relative Paths

Absolute Path → Always appended to your domain

Relative → Appended to prev url

In react-routing, `pathname: "/new-post",`

always refers to absolute path. How to turn this into relative path?

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

14. Absolute vs Relative Paths (Article)

You learned about `<Link>` , you learned about the `to` property it uses.

The path you can use in `to` can be either absolute or relative.

1.1.1.1 Absolute Paths

By default, if you just enter `to="/some-path"` or `to="some-path"` , that's an absolute path.

Absolute path means that it's always appended right after your domain. Therefore, both syntaxes (with and without leading slash) lead to `example.com/some-path` .

1.1.1.2 Relative Paths

Sometimes, you might want to create a relative path instead. This is especially useful, if your component is already loaded given a specific path (e.g. `posts`) and you then want to append something to that existing path (so that you, for example, get `/posts/new`).

If you're on a component loaded via `/posts` , `to="new"` would lead to `example.com/new` , NOT `example.com/posts/new` .

To change this behavior, you have to find out which path you're on and add the new fragment to that existing path. You can do that with the `url` property of `props.match` :

<Link to={props.match.url + '/new'}> will lead to example.com/posts/new when placing this link in a component loaded on /posts . If you'd use the same <Link> in a component loaded via /all-posts , the link would point to /all-posts/new .

There's no better or worse way of creating Link paths - choose the one you need. Sometimes, you want to ensure that you always load the same path, no matter on which path you already are => Use absolute paths in this scenario.

Use relative paths if you want to navigate relative to your existing path.

15. Styling the Active Route

Use a special css class to the active link.

Use 'NavLink' in place of 'Link' in Blog.js. This adds some default styling too.

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

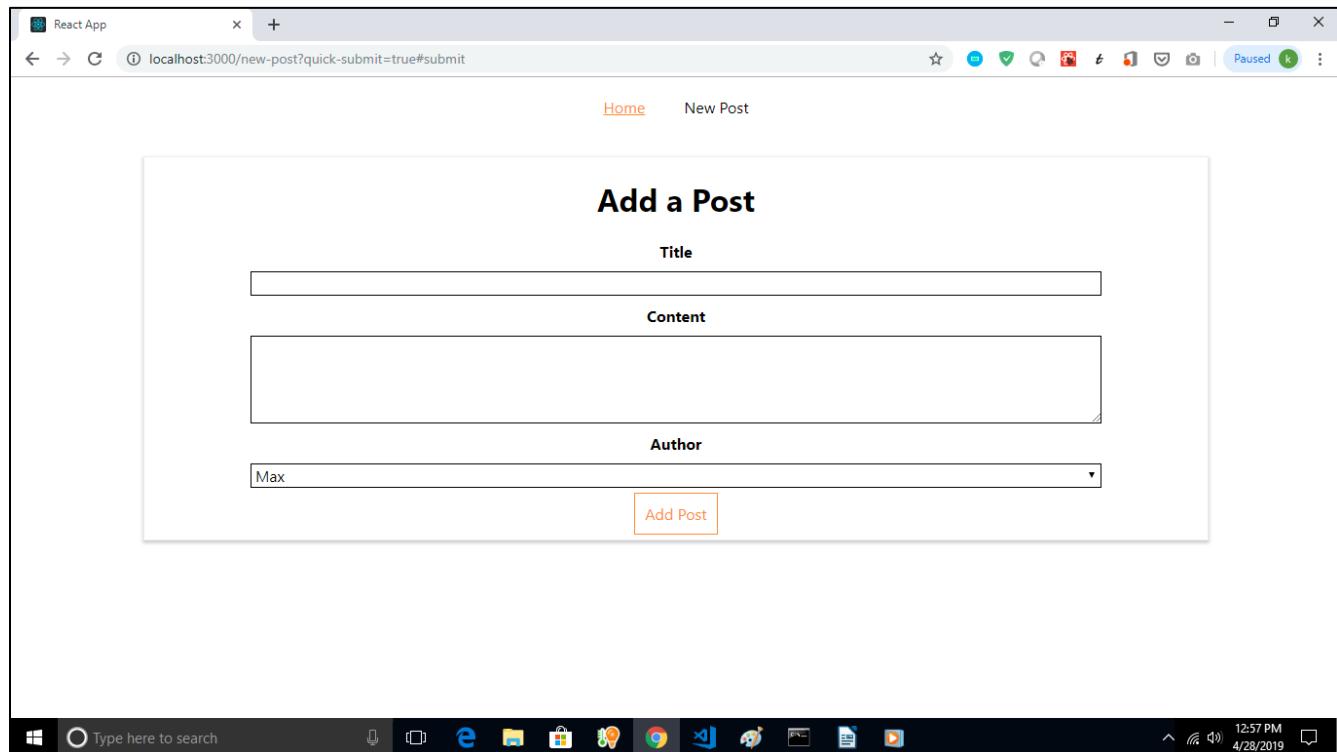
If you do not want to user default 'active' class, you can override this with your own class.

```
<li>
  <NavLink to="/" activeClassName="my-active">
    Home
  </NavLink>
</li>
```

You can also use inline style.

```
<li>
  <NavLink
    to="/"
    activeClassName="my-active"
    activeStyle={{
      color: "#fa923f",
      textDecoration: "underline"
    }}
  >
    Home
  </NavLink>
</li>
```

O/P after inline style:



Blog.js

```
import React, { Component } from "react";
import axios from "../../axios";
import "./Blog.css";
import Posts from "./Posts/Posts";
import { Route, NavLink } from "react-router-dom";
import NewPost from "./NewPost/NewPost";

class Blog extends Component {
  componentDidMount() {
    axios
      .get("/posts")
      .then(response => {
        const posts = response.data.slice(0, 4);
        const updatedPosts = posts.map(post => {
          return {
            ...post,
            author: "Max"
          };
        });
        this.setState({
          posts: updatedPosts
        });
      })
      .catch(error => {
        console.log(error);
      });
  }
}

export default Blog;
```

```

    });
    this.setState({ posts: updatedPosts });
    // console.log( response );
})
.catch(error => {
    // console.log(error);
    this.setState({ error: true });
});
}

render() {
    return (
        <div className="Blog">
            <header>
                <nav>
                    <ul>
                        <li>
                            <NavLink
                                to="/"
                                activeClassName="my-active"
                                activeStyle={{
                                    color: "#fa923f",
                                    textDecoration: "underline"
                                }}
                            >
                                Home
                            </NavLink>
                        </li>
                        <li>
                            <NavLink
                                to={({ pathname: "/new-post",
                                        hash: "#submit", // To jump
                                        to any point
                                        search: "?quick-submit=true"
                                })
                            >
                                New Post
                            </NavLink>
                        </li>
                    </ul>
                </nav>
            </header>
            <main>
                {this.state.error ? <div>Error</div> : <PostList posts={this.state.posts} />}
            </main>
        </div>
    );
}

```

```

</NavLink>
          </li>
        </ul>
      </nav>
    </header>
    {/* <Route path="/" exact render={() =>
<h1>Home</h1>} />
<Route path="/" render={() => <h1>Home 2</h1>} /> */}
      /* this will load Posts component */
      <Route path="/" exact component={Posts} />
      <Route path="/new-post" component={NewPost} />
    </div>
  );
}
}

export default Blog;

```

16. Passing Route Parameters

To click on a single post and load that. Right now after clicking post, nothing happens, the reason being that `clicked={() => this.postSelectedHandler(post.id)}`

is setting some irrelevant state.

Blog.js

```

import React, { Component } from "react";
import axios from "../../axios";
import "./Blog.css";
import Posts from "./Posts/Posts";
import { Route, NavLink } from "react-router-dom";
import NewPost from "./NewPost/NewPost";
import FullPost from "./FullPost/FullPost";

class Blog extends Component {
  componentDidMount() {
    axios
      .get("/posts")

```

```

        .then(response => {
            const posts = response.data.slice(0, 4);
            const updatedPosts = posts.map(post => {
                return {
                    ...post,
                    author: "Max"
                };
            });
            this.setState({ posts: updatedPosts });
            // console.log( response );
        })
        .catch(error => {
            // console.log(error);
            this.setState({ error: true });
        });
    }

    render() {
        return (
            <div className="Blog">
                <header>
                    <nav>
                        <ul>
                            <li>
                                <NavLink
                                    to="/"
                                    activeClassName="my-active"
                                    activeStyle={{
                                        color: "#fa923f",
                                        textDecoration: "underline"
                                    }}>
                                    >
                                    Home
                            </NavLink>
                            </li>
                            <li>
                                <NavLink
                                    to={({ pathname: "/new-post", })

```

```

        hash: "#submit", // To jump
to any point
search: "?quick-submit=true"
// Allows us to use query params
        }
    >
        New Post
</NavLink>
            </li>
        </ul>
    </nav>
</header>
/* <Route path="/" exact render={() =>
<h1>Home</h1>} />
<Route path="/" render={() => <h1>Home 2</h1>} /> */}
/* this will load Posts component */
<Route path="/" exact component={Posts} />
<Route path="/new-post" component={NewPost} />
/* Dynamic id */
<Route path="/:id" exact component={FullPost} />
</div>
);
}
}

export default Blog;

```

Posts.js

```

import React, { Component } from "react";
import axios from "....../axios";
import Post from "....../components/Post/Post";
import "./Posts.css";
import { Link } from "react-router-dom";

class Posts extends Component {
    state = {
        posts: []

```

```

};

componentDidMount() {
  console.log(this.props);
  axios
    .get("/posts")
    .then(response => {
      const posts = response.data.slice(0, 4);
      const updatedPosts = posts.map(post => {
        return {
          ...post,
          author: "Max"
        };
      });
      this.setState({ posts: updatedPosts });
      // console.log( response );
    })
    .catch(error => {
      console.log(error);
      // this.setState({ error: true });
    });
}

postSelectedHandler = id => {
  this.setState({ selectedPostId: id });
};

render() {
  let posts = <p style={{ textAlign: "center" }}>Something went
wrong!</p>;
  if (!this.state.error) {
    posts = this.state.posts.map(post => {
      return (
        <Link to={"/" + post.id} key={post.id}>
          <Post
            title={post.title}
            author={post.author}
            clicked={() =>
this.postSelectedHandler(post.id)}
      
```

```
        />
      </Link>
    );
  });
}

return <section className="Posts">{posts}</section>;
}

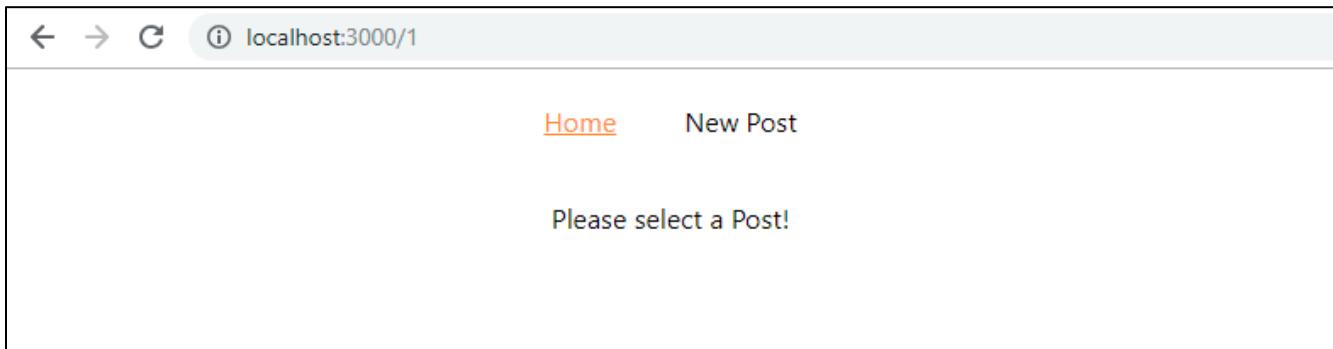
export default Posts;
```

O/P:

The screenshot shows a web browser window at localhost:3000. The page title is "localhost:3000". There are two navigation links: "Home" (underlined) and "New Post". Below them is a grid of four cards, each representing a post:

- Post 1:** Content: **sunt aut facere repellat
provident occaecati
excepturi optio
reprehenderit**. Author: Max.
- Post 2:** Content: **qui est esse**. Author: Max.
- Post 3:** Content: **ea molestias quasi
exercitationem repellat
qui ipsa sit aut**. Author: Max.
- Post 4:** Content: **eum et est occaecati**. Author: Max.

O/P after clicking a post:



17. Extracting Route Parameters

In FullPost.js 'this.props.id' does not work anymore.

Change componentDidUpdate to componentDidMount.

Now it loads the post after clicking it.

O/P:



However there ia a problem. If you click on 'New Post., it also loads the selected post.

The screenshot shows a web application interface. At the top, the URL is `localhost:3000/new-post?quick-submit=true#submit`. Below the URL, there are navigation links: [Home](#) and [New Post](#). The main content area has a title **Add a Post**. It contains three input fields: **Title** (empty), **Content** (empty), and **Author** (a dropdown menu set to `Max`). Below these fields is a button labeled **Add Post**, which is highlighted with an orange border. In the bottom right corner of the main content area, there is a box containing two lines of text in a bold, stylized font: **sunt aut facere repellat provident occaecati
excepturi optio reprehenderit**. Underneath this box, there is some smaller, less prominent text: `quia et suscipit suscipit recusandae consequuntur expedita et cum rephenderit molestiae ut ut
quas totam nostrum rerum est autem sunt rem eveniet architecto`. At the very bottom of the page, there is a small red **Delete** link.

18. Parsing Query Parameters _ the Fragment

You learned how to extract route parameters ($\Rightarrow :id$ etc).

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

1.1.1.3 Query Params:

You can pass them easily like this:

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

or

```
<Link
  to={{ pathname: '/my-path',
        search: '?start=5'
      }}
>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:

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

`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`).

1.1.1.4 Fragment:

You can pass it easily like this:

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

or

```
<Link
  to={{ pathname: '/my-path',
        hash: 'start-position'
      }}
>Go to Start</Link>
```

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

19. Using Switch to Load a Single Route

Blog.js

```
<Route path="/" exact component={Posts} />
  <Route path="/new-post" component={NewPost} />
{ /* Dynamic id */ }
<Route path="/:id" exact component={FullPost} />
```

The problem with above code is that evrrytime all the three Route will be displayed, whenever URL starts with '/'. This can be fixed in the following way:

Blog.js

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

Posts.js

```
return (
  <Link to={"/posts/" + post.id} key={post.id}>
```

This fixes the issue of selected post displayed in the 'New Post' page.

But sometimes you cannot tweak the URL bcz you have a restriction to how theURL should look like. We can still tell react-router to load only one route at one time. This can be achieved by wrapping our Route config with another Component, 'Switch'. Switch tells react-route to only load one of the routes, the first one that matches from given set of routes.

Blog.js

```
import React, { Component } from "react";
import axios from "../../axios";
import "./Blog.css";
import Posts from "./Posts/Posts";
import { Route, NavLink, Switch } from "react-router-dom";
import NewPost from "./NewPost/NewPost";
import FullPost from "./FullPost/FullPost";
```

```
class Blog extends Component {
  componentDidMount() {
    axios
      .get("/posts")
      .then(response => {
        const posts = response.data.slice(0, 4);
        const updatedPosts = posts.map(post => {
          return {
            ...post,
            author: "Max"
          };
        });
        this.setState({ posts: updatedPosts });
        // console.log( response );
      })
      .catch(error => {
        // console.log(error);
        this.setState({ error: true });
      });
  }

  render() {
    return (
      <div className="Blog">
        <header>
          <nav>
            <ul>
              <li>
                <NavLink
                  to="/"
                  activeClassName="my-active"
                  activeStyle={{
                    color: "#fa923f",
                    textDecoration: "underline"
                  }>
                  >
                  Home
                </NavLink>
              
          
        
        <main>
          {this.state.error ? <div>Error</div> : <div>Posts</div>}
        
      </div>
    );
  }
}
```

```

        </li>
        <li>
          <NavLink
            to={{
              pathname: "/new-post",
              hash: "#submit", // To jump
              to any point
              search: "?quick-submit=true"
            }}
          >
            New Post
        </NavLink>
      
```


</nav>

</header>

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

```

<h1>Home</h1>} />
<Route path="/" render={() => <h1>Home 2</h1>} /> */}
    {/* this will load Posts component */}
    <Switch>
      <Route path="/" exact component={Posts} />
      <Route path="/new-post" component={NewPost} />
      {/* Dynamic id */}
      <Route path="/:id" exact component={FullPost} />
    </Switch>
  </div>
);
}
}

export default Blog;

```

Posts.js

```

import React, { Component } from "react";
import axios from "../../../../../axios";

```

```

import Post from "../../components/Post/Post";
import "./Posts.css";
import { Link } from "react-router-dom";

class Posts extends Component {
  state = {
    posts: []
  };

  componentDidMount() {
    console.log(this.props);
    axios
      .get("/posts")
      .then(response => {
        const posts = response.data.slice(0, 4);
        const updatedPosts = posts.map(post => {
          return {
            ...post,
            author: "Max"
          };
        });
        this.setState({ posts: updatedPosts });
        // console.log( response );
      })
      .catch(error => {
        console.log(error);
        // this.setState({ error: true });
      });
  }

  postSelectedHandler = id => {
    this.setState({ selectedPostId: id });
  };

  render() {
    let posts = <p style={{ textAlign: "center" }}>Something went
wrong!</p>;
    if (!this.state.error) {
      posts = this.state.posts.map(post => {

```

```

        return (
          <Link to={"/" + post.id} key={post.id}>
            <Post
              title={post.title}
              author={post.author}
              clicked={() =>
this.postSelectedHandler(post.id)}
            />
          </Link>
        );
      );
    }
    return <section className="Posts">{posts}</section>;
  }
}

export default Posts;

```

20. Navigating Programmatically

IN Posts.js we are using <Link> to return a post. Now, we will remove <Link> and try to achieve the same. Do the following:

- 1) Remove <Link>
- 2) Add key to <Post>

```

return (
  // <Link to={"/" + post.id} key={post.id}>
  <Post
    key={post.id}
    title={post.title}
    author={post.author}
    clicked={() => this.postSelectedHandler(post.id)}
  />
  // </Link>
);

```

Change postSelectedHandler()

```

postSelectedHandler = id => {
  this.props.history.push({ pathname: "/" + id });
  // This will also work.
  // this.props.history.push('/' + id);
};

}

```

21. Additional Information Regarding Active Links

- (1) Change the 'Home' link's name to 'Posts'.
- (2) Highlight 'Posts' when a single post is displayed. Creates edge cases so revert it. Just do the (1)

22. Understanding Nested Routes

Load a component inside another component which is also loaded by routing.

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

from Blog.js and add into Posts.js. You can use `<Route>` component wherever you want in your application as long as page where you are using it is wrapped with `<BrowserRouter>`.

Blog.js

```

import React, { Component } from "react";
import axios from "../../axios";
import "./Blog.css";
import Posts from "./Posts/Posts";
import { Route, NavLink, Switch } from "react-router-dom";
import NewPost from "./NewPost/NewPost";

class Blog extends Component {
  componentDidMount() {
    axios
      .get("/posts")
      .then(response => {
        const posts = response.data.slice(0, 4);
        const updatedPosts = posts.map(post => {
          return {
            ...post,
            author: "Max"
          };
        });
        this.setState({
          posts: updatedPosts
        });
      })
      .catch(error => {
        console.log(error);
      });
  }
}

export default Blog;

```

```

        };
    });
    this.setState({ posts: updatedPosts });
    // console.log( response );
})
.catch(error => {
    // console.log(error);
    this.setState({ error: true });
});
}

render() {
    return (
        <div className="Blog">
            <header>
                <nav>
                    <ul>
                        <li>
                            <NavLink
                                to="/"
                                exact
                                activeClassName="my-active"
                                activeStyle={{
                                    color: "#fa923f",
                                    textDecoration: "underline"
                                }}
                            >
                                Posts
                            </NavLink>
                        </li>
                        <li>
                            <NavLink
                                to={{
                                    pathname: "/new-post",
                                    hash: "#submit", // To jump
to any point
                                    search: "?quick-submit=true"
                                }}
                            // Allows us to use query params
                        </li>
                    </ul>
                </nav>
            </header>
            <main>
                {this.state.error ? <h1>Error</h1> : <PostList posts={this.state.posts} />}
            </main>
        </div>
    );
}

```

```

        >
        New Post
</NavLink>
            </li>
        </ul>
    </nav>
</header>
/* <Route path="/" exact render={() =>
<h1>Home</h1>} />
<Route path="/" render={() => <h1>Home 2</h1>} /> */}
/* this will load Posts component */
<Switch>
    <Route path="/new-post" component={NewPost} />
    <Route path="/" component={Posts} />
</Switch>
</div>
);
}
}

export default Blog;

```

Posts.js

```

import React, { Component } from "react";
import axios from "../../../../../axios";
import Post from "../../../../../components/Post/Post";
import "./Posts.css";
import { Route } from "react-router-dom";
import FullPost from "../FullPost/FullPost";

class Posts extends Component {
    state = {
        posts: []
    };

    componentDidMount() {
        console.log(this.props);
    }
}

export default Posts;

```

```

axios
  .get("/posts")
  .then(response => {
    const posts = response.data.slice(0, 4);
    const updatedPosts = posts.map(post => {
      return {
        ...post,
        author: "Max"
      };
    });
    this.setState({ posts: updatedPosts });
    // console.log( response );
  })
  .catch(error => {
    console.log(error);
    // this.setState({ error: true });
  });
}

postSelectedHandler = id => {
  this.props.history.push({ pathname: "/" + id });
  // This will also work.
  // this.props.history.push('/' + id);
};

render() {
  let posts = <p style={{ textAlign: "center" }}>Something went
wrong!</p>;
  if (!this.state.error) {
    posts = this.state.posts.map(post => {
      return (
        // <Link to={"/" + post.id} key={post.id}>
        <Post
          key={post.id}
          title={post.title}
          author={post.author}
          clicked={() =>
this.postSelectedHandler(post.id)}
        />
      );
    });
  }
  return (
    <div style={{ width: "100%", padding: "20px" }}>
      {posts}
    </div>
  );
}

```

```

        // </Link>
    );
}
return (
<div>
    <section className="Posts">{posts}</section>
    {/* If 'exact' is used here, selected post will not
be displayed*/}
        {/* If url is changed for any reason it will
automatically pick that */}
        {/* <Route path={this.props.match.url + "/:id"}>
component={FullPost} /> */}
        <Route path="/:id" component={FullPost} />
    </div>
);
}
}

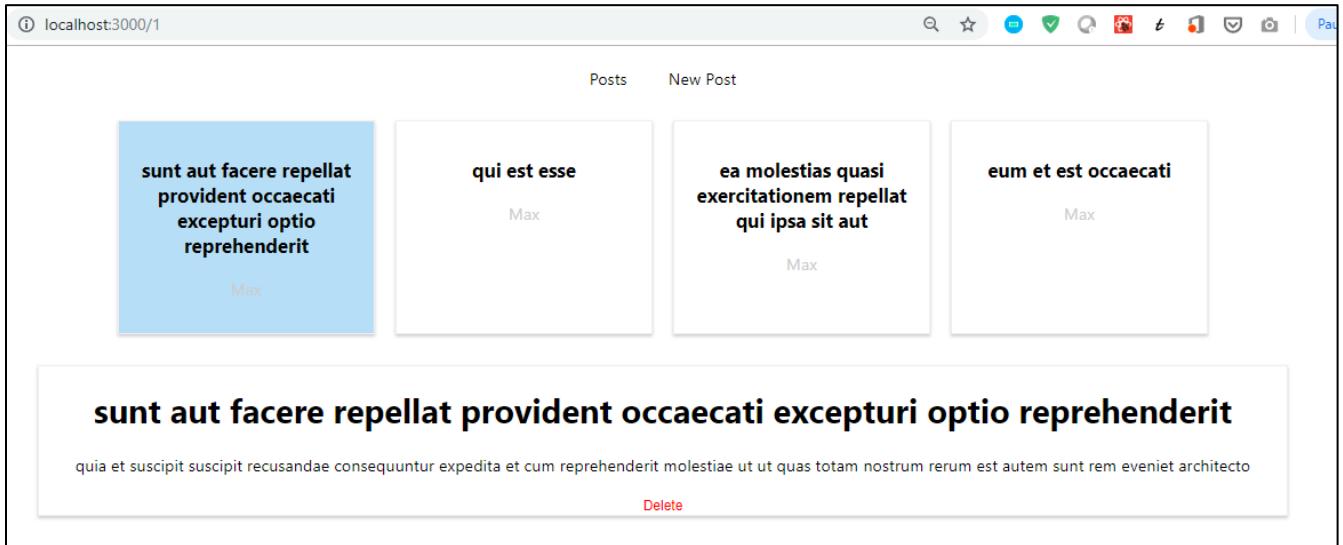
export default Posts;

```

O/P:



O/P after clicking on any post:



23. Creating Dynamic Nested Routes

In the prev step, if we click on a post after already selecting a post, URL changes but the post does not. The reason is that react-router does not replace component each time.

Implement componentDidUpdate() in FullPost.js to achieve this.

FullPost.js

```
import React, { Component } from "react";
import axios from "../../../../../axios";
import "./FullPost.css";

class FullPost extends Component {
  state = {
    loadedPost: null
  };

  componentDidMount() {
    console.log(this.props.id);
    this.loadData();
  }
  componentDidUpdate() {
    this.loadData();
  }

  loadData() {
    if (this.props.match.params.id) {
      axios.get(`https://jsonplaceholder.typicode.com/posts/${this.props.match.params.id}`)
        .then(res => {
          this.setState({loadedPost: res.data});
        })
        .catch(error => {
          console.log(error);
        });
    }
  }
}

export default FullPost;
```

```

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

    deletePostHandler = () => {
        axios.delete("/posts/" +
this.props.match.params.id).then(response => {
            console.log(response);
        });
    };
}

render() {
    let post = <p style={{ textAlign: "center" }}>Please select a
Post!</p>;
    if (this.props.match.params.id) {
        post = <p style={{ textAlign: "center"
}}>Loading...!</p>;
    }
    if (this.state.loadedPost) {
        console.log(this.state.loadedPost);
        post = (
            <div className="FullPost">
                <h1>{this.state.loadedPost.title}</h1>
                <p>{this.state.loadedPost.body}</p>
                <div className="Edit">
                    <button onClick={this.deletePostHandler}
className="Delete">
                        Delete
</button>

```

```
        </div>
    </div>
);
}
return post;
}

export default FullPost;
```

24. Redirecting Requests

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

Blog.js inside <Switch>

```
<Redirect from="/" to="/posts"/>
```

25. Conditional Redirects

NewPost.js

```
import React, { Component } from "react";
import axios from "../../../../../axios";
import { Redirect } from "react-router-dom";
import "./NewPost.css";

class NewPost extends Component {
  state = {
    title: "",
    content: "",
    author: "Max",
    submitted: false
  };

  postDataHandler = () => {
    const data = {
      title: this.state.title,
```

```

        body: this.state.content,
        author: this.state.author
    };
    axios.post("/posts", data).then(response => {
        console.log(response);
        this.setState({ submitted: true });
    });
};

render() {
    let redirect = null;
    if (this.state.submitted) {
        redirect = <Redirect to="/posts" />;
    }
    return (
        <div className="NewPost">
            {redirect}
            <h1>Add a Post</h1>
            <label>Title</label>
            <input
                type="text"
                value={this.state.title}
                onChange={event => this.setState({ title:
event.target.value })}
            />
            <label>Content</label>
            <textarea
                rows="4"
                value={this.state.content}
                onChange={event => this.setState({ content:
event.target.value })}
            />
            <label>Author</label>
            <select
                value={this.state.author}
                onChange={event => this.setState({ author:
event.target.value })}>
                <option value="Max">Max</option>

```

```

        <option value="Manu">Manu</option>
    </select>
    <button onClick={this.postDataHandler}>Add
Post</button>
    </div>
)
}
}

export default NewPost;

```

26. Using the History Prop to Redirect (Replace)

NewPost.js

```

axios.post("/posts", data).then(response => {
  console.log(response);
  this.props.history.replace("/posts");
//this.props.history.push('/posts');
  // this.setState({ submitted: true });
});

```

Redirect replaces the current page, whereas push/replace pushes the current page into the stack so that you can go back.

27. Working with Guards

Navigation Guards

When you do not know that the user is authenticated or not, there are some part/routes in the application that you only want to allow user to visit if he is authenticated.

In Blog.js, we want to show the below only when the user is authenticated.

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

Create an auth state and assign false initially.

Blog.js

```
import React, { Component } from "react";
import axios from "../../axios";
import "./Blog.css";
import Posts from "./Posts/Posts";
import { Route, NavLink, Switch, Redirect } from "react-router-dom";
import NewPost from "./NewPost/NewPost";

class Blog extends Component {
  state = {
    auth: false
  }
  componentDidMount() {
    axios
      .get("/posts")
      .then(response => {
        const posts = response.data.slice(0, 4);
        const updatedPosts = posts.map(post => {
          return {
            ...post,
            author: "Max"
          };
        });
        this.setState({ posts: updatedPosts });
        // console.log( response );
      })
      .catch(error => {
        // console.log(error);
        this.setState({ error: true });
      });
  }

  render() {
    return (
      <div className="Blog">
        <header>
          <nav>
            <ul>
              <li>
                <NavLink
              </ul>
            </nav>
          </header>
        <div>
```

```

        to="/" 
        exact
        activeClassName="my-active"
        activeStyle={{
            color: "#fa923f",
            textDecoration: "underline"
        }}
    >
    Posts
</NavLink>
</li>
<li>
    <NavLink
        to={{{
            pathname: "/new-post",
            hash: "#submit", // To jump to any point
            search: "?quick-submit=true" // Allows us to use
query params
        }}>
    >
        New Post
</NavLink>
</li>
</ul>
</nav>
</header>
{/* <Route path="/" exact render={() => <h1>Home</h1>} />
<Route path="/" render={() => <h1>Home 2</h1>} /> */}
/* this will load Posts component */
/* The below is a guard. An alternative to this would
be to go the guarded page and write auth condition in
componentDidMount() */
<Switch>
    {this.state.auth ? <Route path="/new-post"
component={NewPost} /> : null}
        <Route path="/posts" component={Posts} />
        <Redirect from="/" to="/posts"></Redirect>
    </Switch>
</div>

```

```

        );
    }
}

export default Blog;

```

O/P: When clicking on the ‘New Posts’, we are redirected to ‘Posts’ as user is unauthenticated and cannot access ‘New Posts’.

28. Handling the 404 Case (Unknown Routes)

Blog.js

```

<Switch>
  {this.state.auth ? <Route path="/new-post"
component={NewPost} /> : null}
  <Route path="/posts" component={Posts} />
  {/* This will catch all the unknown routes.
  This will not work together with <Redirect>, if you redirect
from="/" bcz
  from="/" is treated as prefix therefore this catches all
routes
  as does <Route render={() => <h1>not Found</h1>} />.  */}
  <Route render={() => <h1>not Found</h1>} />
  {/* <Redirect from="/" to="/posts"></Redirect> */}
</Switch>

```

29. Loading Routes Lazily

30. Routing and Server Deployment

31. Time to Practice – Routing

32. Wrap Up

33. Useful Resources _ Links

12. Adding Routing to our Burger Project

13. Forms and Form Validation

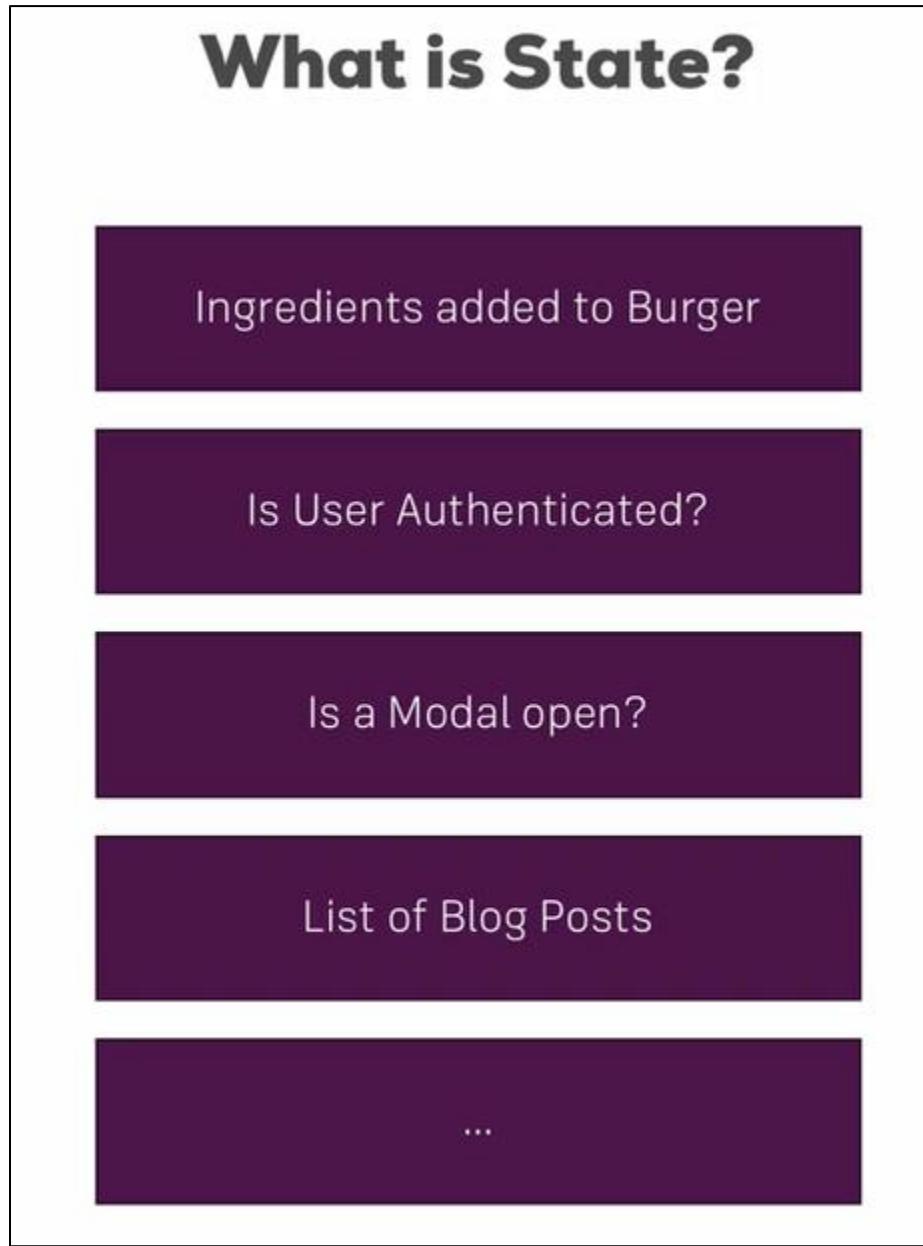
1. Module Introduction
2. Analyzing the App
3. Creating a Custom Dynamic Input Component
4. Setting Up a JS Config for the Form
5. Dynamically Create Inputs based on JS Config
6. Adding a Dropdown Component
7. Handling User Input
8. Handling Form Submission
9. Adding Custom Form Validation
10. Fixing a Common Validation Gotcha
11. Adding Validation Feedback
12. Improving Visual Feedback
13. Showing Error Messages
14. Handling Overall Form Validity
15. Working on an Error
16. Fixing a Bug

14. Redux

1. Module Introduction

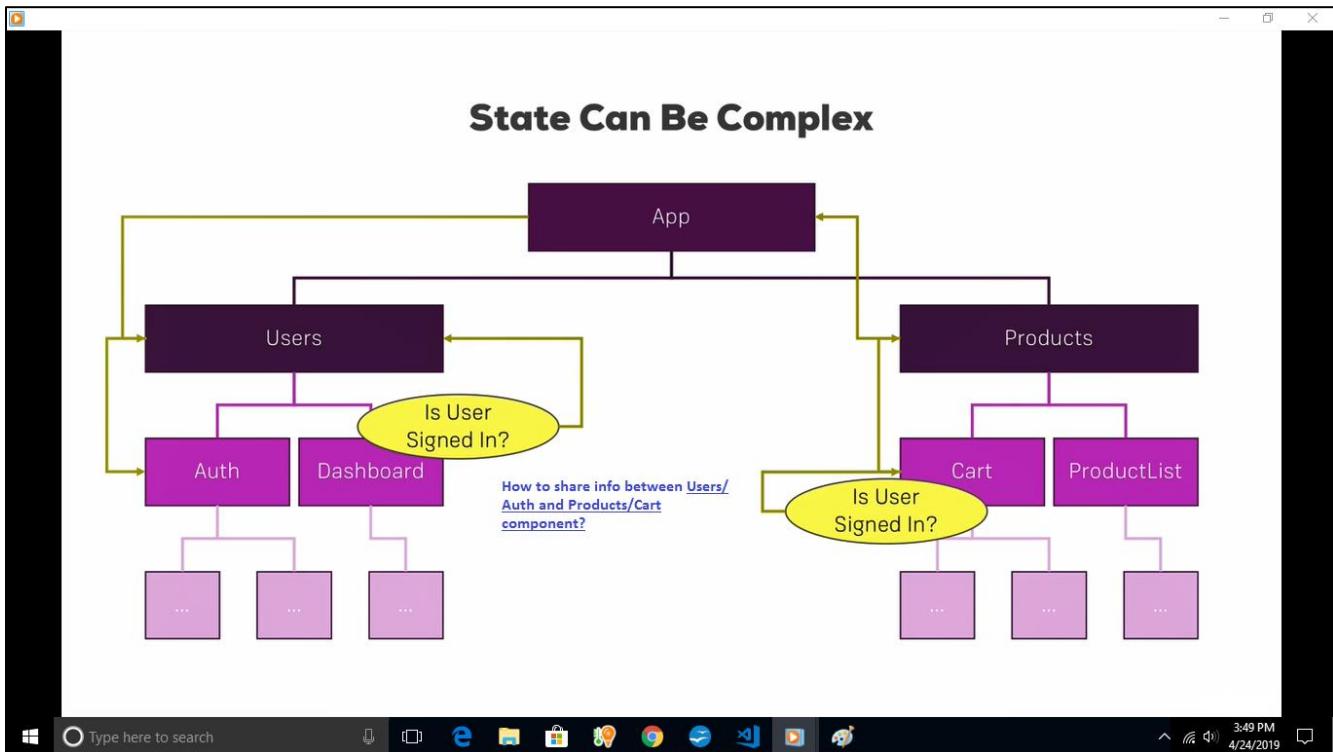
To manage state of the application easier. Redux is a 3rd party library that works independent of react.

2. Understanding State



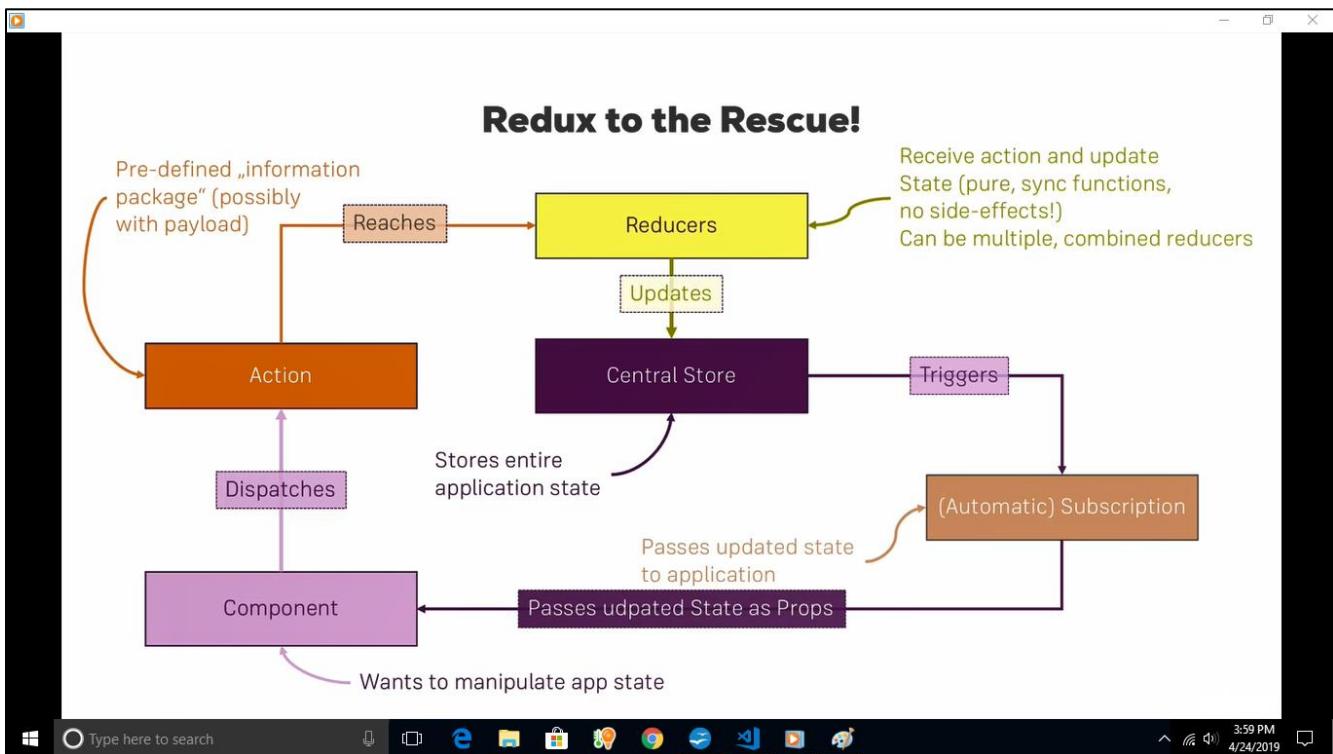
3. The Complexity of Managing State

React has built in 'state' property but passing ingredients (Burger project) from component A to component B is difficult, and we had to use routing query params (not a very elegant solution) for that.



React's reactivity system does not respond to changes in global variables. So Setting global variables/states for common requirements across the application will not work.

4. Understanding the Redux Flow



5. Setting Up Reducer and Store

```
>> npm install --save redux
```

Create a file → redux-basics.js → in root folder. Outside react directory. This file will be run with node to show independence of redux from react.

q) How to run a javascript file in node?

```
D:\react-redux> node redux-basics.js
```

redux-basics.js

```
// Node style of import
const redux = require("redux");
// createStore is a function, but do not execute it yet
const createStore = redux.createStore;

const initialState = {
  counter: 0
};

// Reducer
// state --> current state
// Function has to return the updated state
// 'state = initialState' --> Take initialState when state is
undefined
const rootReducer = (state = initialState, action) => {
  return state; // just returning the current state
};

// Store

const store = createStore(rootReducer);
console.log(store.getState()); // { counter: 0 }
/*
A store needs to be initialized with a reducer.
We have one reducer even if we combine multiple ones, they will be
merged into one.
Reducer is strongly connected to the store. it is the only thing that
can update the state.
We need to pass reducer in the common function above.
```

```
/*
// Dispatching Action
// Subscription
```

6. Dispatching Action

redux-basics.js

```
// Node style of import
const redux = require("redux");
// createStore is a function, but do not execute it yet
const createStore = redux.createStore;

const initialState = {
    counter: 0
};

// Reducer
// state --> current state
// Function has to return the updated state
// 'state = initialState' --> Take initialState when state is
undefined
const rootReducer = (state = initialState, action) => {
    if (action === "INC_COUNTER") {
        return {
            ...state,
            counter: state.counter + 1
        };
    }
    if (action === "ADD_COUNTER") {
        return {
            ...state,
            counter: state.counter + action.value
        };
    }
    return state; // just returning the current state
};

// Store
```

```

const store = createStore(rootReducer);
console.log(store.getState()); // { counter: 0 }
/*
A store needs to be initialized with a reducer.
We have one reducer even if we combine multiple ones, they will be
merged into one.
Reducer is strongly connected to the store. it is the only thing that
can update the state.
We need to pass reducer in the common function above.
*/
// Dispatching Action
// You have to pass 'type' property and the value has to be in uppercase.

store.dispatch({ type: "INC_COUNTER" });
store.dispatch({ type: "ADD_COUNTER", value: 10 });
console.log(store.getState());

// Subscription

```

Expected o/p:

```

PS D:\react-redux> node redux-basics.js
{ counter: 0 }
{ counter:11 }

```

7. Adding Subscriptions

redux-basics.js

```

// Node style of import
const redux = require("redux");
// createStore is a function, but do not execute it yet
const createStore = redux.createStore;

const initialState = {
  counter: 0
};

```

```

// Reducer
// state --> current state
// Function has to return the updated state
// 'state = initialState' --> Take initialState when state is
// undefined
const rootReducer = (state = initialState, action) => {
  if (action === "INC_COUNTER") {
    return {
      ...state,
      counter: state.counter + 1
    };
  }
  if (action === "ADD_COUNTER") {
    return {
      ...state,
      counter: state.counter + action.value
    };
  }
  return state; // just returning the current state
};

// Store

const store = createStore(rootReducer);
console.log(store.getState()); // { counter: 0 }
/*
A store needs to be initialized with a reducer.
We have one reducer even if we combine multiple ones, they will be
merged into one.
Reducer is strongly connected to the store. it is the only thing that
can update the state.
We need to pass reducer in the common function above.
*/
// Subscription
/*
subscription makes sure that you do not have to call getState()
manually.

```

```

It will inform when do I need to get state.
the code inside subscribe will be executed whenever action is
dispatched or state is changed.

*/
store.subscribe(() => {
  console.log("[subscription]", store.getState());
});

// Dispatching Action
// You have to pass 'type' property and the value has to be in upper
// case.

store.dispatch({ type: "INC_COUNTER" });
store.dispatch({ type: "ADD_COUNTER", value: 10 });
console.log(store.getState());

```

Expected O/P:

PS D:\react-redux> node redux-basics.js

```

{ counter: 0 }

[subscription] { counter: 1 }

[subscription] { counter: 11 }

{ counter: 11 }

```

8. Connecting React to Redux

The store should be created right before our application starts.

Create 'store' folder next to the containers. Create 'reducer.js' in store folder. This will now be the file we will export for the reducer we want to use.

Import this 'reducer.js' into index.js. Pass this reducer to createStore() in index.js.

Index.js

```

import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
import * as serviceWorker from "./serviceWorker";

```

```

import { createStore } from "redux";
import reducer from "./store/reducer";

// This takes a reducer as an input.
/* We will have more complex reducers in our application with lot of
code for different types of actions, so we typically store their
logics into their own files
*/

const store = createStore(reducer);

ReactDOM.render(<App />, document.getElementById("root"));

// If you want your app to work offline and load faster, you can
// change
// unregister() to register() below. Note this comes with some
// pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();

```

reducer.js

```

const initialState = {
  counter: 0
};

const reducer = (state = initialState, action) => {
  return state;
};
export default reducer;

```

We also need to connect store to our react application. Next lecture.

9. Connecting the Store to React

To connect redux to react, we need to install a package. This package allows to hook up redux store to react application.

>> npm install --save react-redux

In Index.js

1) Import {Provider} from react-redux

```
import { Provider } from "react-redux";
```

2) Wrap <App> component with <Provider>

```
ReactDOM.render(  
  <Provider>  
    <App />  
  </Provider>,  
  document.getElementById("root")  
);
```

Providers are helpers components which allows us to inject our store into react components.

3) To hook up provider component with the store, set up 'store' property in <Provider> and provide the value as the store created with createStore(). With this, store is somewhat connected to react application.

```
const store = createStore(reducer);  
  
ReactDOM.render(  
  <Provider store={store}>  
    <App />  
  </Provider>,  
  document.getElementById("root")  
);
```

4) How do we get the data from the global store managed by redux (for ex: counter from reducer.js) in our Counter container?

Ans: Connect Counter container with store. Create a subscription in Counter.js. Use a feature provided by react-redux package to achieve this.

```
import { connect } from "react-redux";
```

connect is a function that returns a higher order component (to be used in export statement).

2 info to be passed to connect(). 1st, which slice of state do I want to get in this container? 2nd, What actions do I want to dispatch?

```
// get state and map it to props
const mapStateToProps = state => {
  return {
    ctr: state.counter
  };
};

export default connect(mapStateToProps)(Counter);
```

Index.js

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
import * as serviceWorker from "./serviceWorker";
import { createStore } from "redux";
import reducer from "./store/reducer";
import { Provider } from "react-redux";

// This takes a reducer as an input.
/*
We will have more complex reducers in our application with lot of
code for different
types of actions, so we typically store their logics into their own
files
*/

const store = createStore(reducer);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

```
// If you want your app to work offline and load faster, you can
// change
// unregister() to register() below. Note this comes with some
// pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Counter.js

```
import React, { Component } from "react";
import { connect } from "react-redux";
import CounterControl from
"../../components/CounterControl/CounterControl";
import CounterOutput from
"../../components/CounterOutput/CounterOutput";

class Counter extends Component {
  state = {
    counter: 0
  };

  counterChangedHandler = (action, value) => {
    switch (action) {
      case "inc":
        this.setState(prevState => {
          return { counter: prevState.counter + 1 };
        });
        break;
      case "dec":
        this.setState(prevState => {
          return { counter: prevState.counter - 1 };
        });
        break;
      case "add":
        this.setState(prevState => {
          return { counter: prevState.counter + value };
        });
        break;
      case "sub":
        this.setState(prevState => {
          return { counter: prevState.counter - value };
        });
        break;
    }
  }
}
```

```

        this.setState(prevState => {
            return { counter: prevState.counter - value };
        });
        break;
    }
};

render() {
    return (
        <div>
            {/* <CounterOutput value={this.state.counter} /> */}
            {/* Change the above statement to the below to get
counter from global state*/}
            <CounterOutput value={this.props.ctr} />
            <CounterControl
                label="Increment"
                clicked={() => this.counterChangedHandler("inc")}
            />
            <CounterControl
                label="Decrement"
                clicked={() => this.counterChangedHandler("dec")}
            />
            <CounterControl
                label="Add 5"
                clicked={() => this.counterChangedHandler("add",
5)}
            />
            <CounterControl
                label="Subtract 5"
                clicked={() => this.counterChangedHandler("sub",
5)}
            />
        </div>
    );
}
// get state and map it to props
const mapStateToProps = state => {
    return {

```

```

        ctr: state.counter
    };
};

export default connect(mapStateToProps)(Counter);

```

10. Dispatching Actions from within the Component

What kind of actions do I want to dispatch to container?

Counter.js

```

import React, { Component } from "react";
import { connect } from "react-redux";
import CounterControl from
"../../components/CounterControl/CounterControl";
import CounterOutput from
"../../components/CounterOutput/CounterOutput";

class Counter extends Component {
  state = {
    counter: 0
  };

  counterChangedHandler = (action, value) => {
    switch (action) {
      case "inc":
        this.setState(prevState => {
          return { counter: prevState.counter + 1 };
        });
        break;
      case "dec":
        this.setState(prevState => {
          return { counter: prevState.counter - 1 };
        });
        break;
      case "add":
        this.setState(prevState => {
          return { counter: prevState.counter + value };
        });
    }
  }
}

export default connect(mapStateToProps)(Counter);

```

```

        break;
    case "sub":
        this.setState(prevState => {
            return { counter: prevState.counter - value };
        });
        break;
    }
};

render() {
    return (
        <div>
            {/* <CounterOutput value={this.state.counter} /> */}
            {/* Change the above statement to the below to get
counter from global state*/}
            <CounterOutput value={this.props.ctr} />
            <CounterControl
                label="Increment"
                // clicked={() =>
this.counterChangedHandler("inc")}
                clicked={() => this.props.onIncrementCounter}
            />
            <CounterControl
                label="Decrement"
                clicked={() => this.counterChangedHandler("dec")}
            />
            <CounterControl
                label="Add 5"
                clicked={() => this.counterChangedHandler("add",
5)}
            />
            <CounterControl
                label="Subtract 5"
                clicked={() => this.counterChangedHandler("sub",
5)}
            />
        </div>
    );
}

```

```

}

// get state and map it to props
const mapStateToProps = state => {
  return {
    ctr: state.counter
  };
};

const mapDispatchToProps = dispatch => {
  return {
    onIncrementCounter: () => dispatch({ type: "INCREMENT" })
  };
};

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter);

```

reducer.js

```

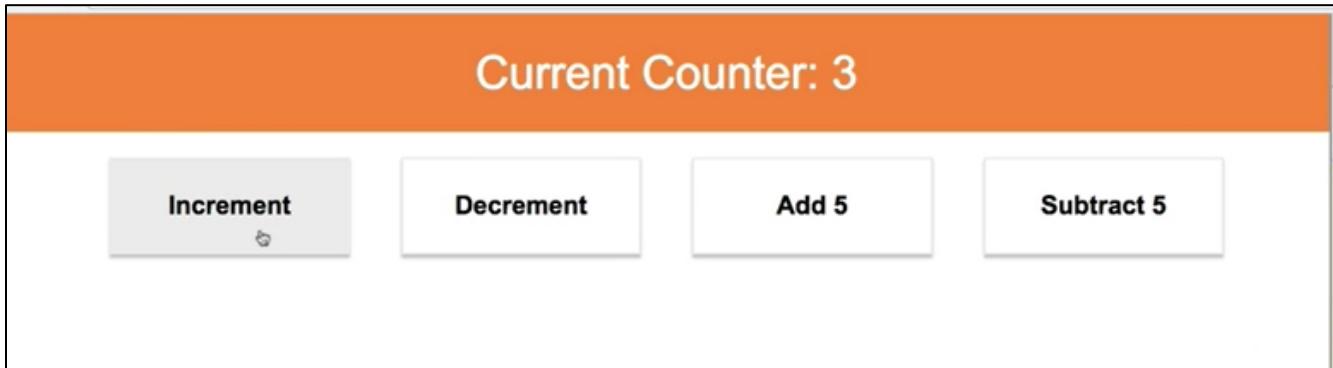
const initialState = {
  counter: 0
};

const reducer = (state = initialState, action) => {
  if (action.type === "INCREMENT") {
    return {
      counter: state.counter + 1
    };
  }
  return state;
};

export default reducer;

```

Note: Increment should increase 'Current Counter'. It is not working. Fix this.



11. Time to Practice - Dispatching Actions

12. Passing and Retrieving Data with Action

Get some data/value with action.

Counter.js

```
const mapDispatchToProps = dispatch => {
  return {
    // Any no of parameters can be passed. Only the 'type' is
    // mandatory
    onIncrementCounter: () => dispatch({ type: "INCREMENT", val:
    10 })
  };
};
```

reducer.js

```
const initialState = {
  counter: 0
};

const reducer = (state = initialState, action) => {
  if (action.type === "INCREMENT") {
    return {
      counter: state.counter + action.val
    };
  }
  return state;
};
```

```
export default reducer;
```

13. Switch-Case in the Reducer

Replace if statements with switch in reducer.js.

14. Updating State Immutable

1. Add a new property 'results' in state of reducer.js.
2. Add a button in Counter.js . onClick of 'store result' button, the current counter should be stored in the . Also when we click on the , the item should be removed from the array.

Reducer.js

```
const initialState = {
  counter: 0,
  results: []
};

const reducer = (state = initialState, action) => {
  if (action.type === "INCREMENT") {
    return {
      ...state,
      counter: state.counter + action.val
    };
  }
  if (action.type === "STORE_RESULT") {
    return {
      ...state,
      results: state.results.concat({ id: new Date(), value: state.counter })
    };
  }
  if (action.type === "DELETE_RESULT") {
    return {
      ...state,
      results: state.results.concat(state.counter)
    };
  }
};
```

```
}

return state;
};

export default reducer;
```

Counter.js

```
import React, { Component } from "react";
import { connect } from "react-redux";
import CounterControl from
"../../components/CounterControl/CounterControl";
import CounterOutput from
"../../components/CounterOutput/CounterOutput";

class Counter extends Component {
  state = {
    counter: 0
  };

  counterChangedHandler = (action, value) => {
    switch (action) {
      case "inc":
        this.setState(prevState => {
          return { counter: prevState.counter + 1 };
        });
        break;
      case "dec":
        this.setState(prevState => {
          return { counter: prevState.counter - 1 };
        });
        break;
      case "add":
        this.setState(prevState => {
          return { counter: prevState.counter + value };
        });
        break;
    }
  }
}
```

```

        case "sub":
            this.setState(prevState => {
                return { counter: prevState.counter - value };
            });
            break;
    }
};

render() {
    return (
        <div>
            {/* <CounterOutput value={this.state.counter} /> */}
            {/* Change the above statement to the below to get counter
from global state*/}
            <CounterOutput value={this.props.ctr} />
            <CounterControl
                label="Increment"
                //    clicked={() => this.counterChangedHandler("inc")}
                clicked={this.props.onIncrementCounter}
            />
            <CounterControl
                label="Decrement"
                clicked={() => this.counterChangedHandler("dec")}
            />
            <CounterControl
                label="Add 5"
                clicked={() => this.counterChangedHandler("add", 5)}
            />
            <CounterControl
                label="Subtract 5"
                clicked={() => this.counterChangedHandler("sub", 5)}
            />

            <hr />
            <button onClick={() =>
this.props.onStoreResult(this.props.ctr)}>
                Store Result
            </button>
            <ul>

```

```

        {this.props.storedResults.map(strResult => (
          <li
            key={strResult.id}
            onClick={() => this.props.onDeleteResult(strResult.id)}
          >
            {strResult.value}
          </li>
        )));
      </ul>
    </div>
  );
}

// get state and map it to props
const mapStateToProps = state => {
  return {
    ctr: state.counter,
    storedResults: state.results
  };
};

const mapDispatchToProps = dispatch => {
  return {
    // Any no of parameters can be passed. Only the 'type' is mandatory
    onIncrementCounter: () => dispatch({ type: "INCREMENT", val: 10 }),
    onStoreResult: result => dispatch({ type: "STORE_RESULT", result: result }),
    onDeleteResult: id => dispatch({ type: "DELETE_RESULT", resultElId: id })
  };
};

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter);

```

O/P:

Current Counter: 0

Increment **Decrement** **Add 5**

Subtract 5

Store Result

This is a screenshot of a user interface for a counter application. The interface is contained within a rectangular frame. At the top, there is an orange header bar with the text "Current Counter: 0" centered in white. Below the header are four rectangular buttons arranged horizontally: "Increment", "Decrement", "Add 5", and "Subtract 5", all in bold black text. Below these buttons is a large, empty rectangular area. At the bottom of the frame is a light gray footer bar with a single button labeled "Store Result".

O/P (After clicking Increment):

Current Counter: 10

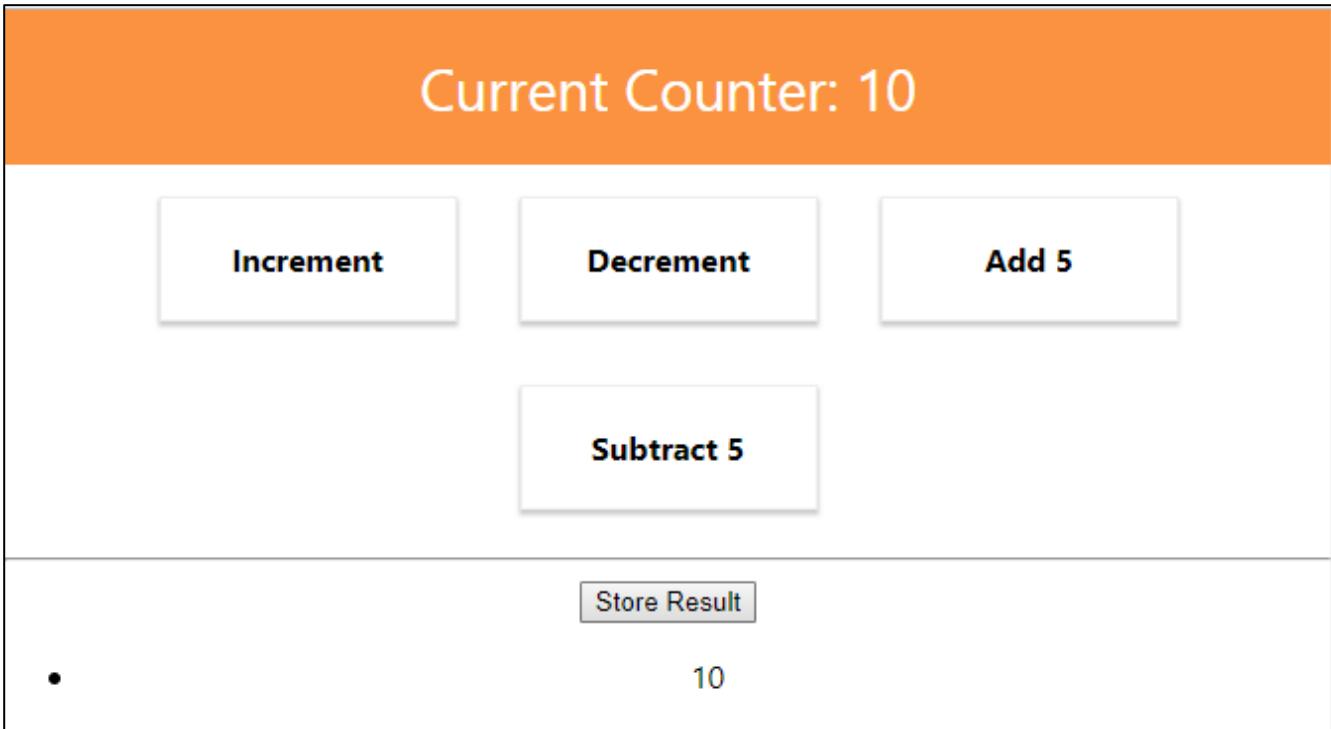
Increment **Decrement** **Add 5**

Subtract 5

Store Result

This is a screenshot of a user interface for a counter application, similar to the one above but with a different value. The interface is contained within a rectangular frame. At the top, there is an orange header bar with the text "Current Counter: 10" centered in white. Below the header are four rectangular buttons arranged horizontally: "Increment", "Decrement", "Add 5", and "Subtract 5", all in bold black text. Below these buttons is a large, empty rectangular area. At the bottom of the frame is a light gray footer bar with a single button labeled "Store Result".

O/P (After clicking Store Result):



Deleting onclick will be taken in the next lecture.

15. Updating Arrays Immutable

There are different ways to delete an item from an array:

- 1) Get the id of the element to be deleted and use splice() method.

State.results.splice(id,1); → 2nd arg indicates the no of elements we want to delete

But this operation is not immutable.

- 2) Create a new array and distribute all the elements of array into new array

```
Const id=2;
```

```
const newArray= [...state.results]; → this creates a copy of the results array
```

```
newArray.splice(id,1);
```

```
return{
```

```
...state,
```

```
Results:newArray
```

```
}
```

Note: if the copy of results array were objects, the objects themselves are still pointing to the same object as it did before. So if you change a property in one of the elements themselves just creating a new array like the above is not enough. If you are just planning to remove an object, then it is OK, bcz you do not touch the object, you just remove it from the array.

- 3) Using filter() -> filter does not touch the old array, rather creates a new one

```
const updatedArray = state.results.filter(result => result.id !== action.resultElId);
```

```
if (action.type === "DELETE_RESULT") {
  const updatedArray = state.results.filter(result => result.id !==
action.resultElId);
  return {
    ...state,
    results: updatedArray
  };
}
```

Reducer.js

```
const initialState = {
  counter: 0,
  results: []
};

const reducer = (state = initialState, action) => {
  if (action.type === "INCREMENT") {
    return {
      ...state,
      counter: state.counter + action.val
    };
  }
  if (action.type === "STORE_RESULT") {
    return {
      ...state,
      results: state.results.concat({ id: new Date(), value:
state.counter })
    };
  }
  if (action.type === "DELETE_RESULT") {
```

```

    const updatedArray = state.results.filter(result => result.id !==
action.resultElId);
    return {
      ...state,
      results: updatedArray
    };
}

return state;
};

export default reducer;

```

Counter.js

```

import React, { Component } from "react";
import { connect } from "react-redux";
import CounterControl from
"../../components/CounterControl/CounterControl";
import CounterOutput from
"../../components/CounterOutput/CounterOutput";

class Counter extends Component {
  state = {
    counter: 0
  };

  counterChangedHandler = (action, value) => {
    switch (action) {
      case "inc":
        this.setState(prevState => {
          return { counter: prevState.counter + 1 };
        });
        break;
      case "dec":
        this.setState(prevState => {
          return { counter: prevState.counter - 1 };
        });
    }
  }
}

export default Counter;

```

```
        break;
    case "add":
        this.setState(prevState => {
            return { counter: prevState.counter + value };
        });
        break;
    case "sub":
        this.setState(prevState => {
            return { counter: prevState.counter - value };
        });
        break;
    }
};

render() {
    return (
        <div>
            {/* <CounterOutput value={this.state.counter} /> */}
            {/* Change the above statement to the below to get counter
from global state*/}
            <CounterOutput value={this.props.ctr} />
            <CounterControl
                label="Increment"
                // clicked={() => this.counterChangedHandler("inc")}
                clicked={this.props.onIncrementCounter}
            />
            <CounterControl
                label="Decrement"
                clicked={() => this.counterChangedHandler("dec")}
            />
            <CounterControl
                label="Add 5"
                clicked={() => this.counterChangedHandler("add", 5)}
            />
            <CounterControl
                label="Subtract 5"
                clicked={() => this.counterChangedHandler("sub", 5)}
            />
    );
}
```

```

        <hr />
        <button onClick={() =>
this.props.onStoreResult(this.props.ctr)}>
            Store Result
        </button>
        <ul>
            {this.props.storedResults.map(strResult => (
                <li
                    key={strResult.id}
                    onClick={() => this.props.onDeleteResult(strResult.id)}
                >
                    {strResult.value}
                </li>
            ))}
        </ul>
    </div>
);
}
}

// get state and map it to props
const mapStateToProps = state => {
    return {
        ctr: state.counter,
        storedResults: state.results
    };
};

const mapDispatchToProps = dispatch => {
    return {
        // Any no of parameters can be passed. Only the 'type' is
        // mandatory
        onIncrementCounter: () => dispatch({ type: "INCREMENT", val: 10 }),
        onStoreResult: result => dispatch({ type: "STORE_RESULT", result: result }),
        onDeleteResult: id => dispatch({ type: "DELETE_RESULT", resultElId: id })
    };
};

```

```
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter);
```

16. Immutable Update Patterns

Immutable Update Patterns on

reduxjs.org: <http://redux.js.org/docs/recipes/reducers/ImmutableUpdatePatterns.html>

1.2 UPDATING NESTED OBJECTS

The key to updating nested data is **that every level of nesting must be copied and updated appropriately**. This is often a difficult concept for those learning Redux, and there are some specific problems that frequently occur when trying to update nested objects. These lead to accidental direct mutation, and should be avoided.

1.2.1.1.1 Common Mistake #1: New variables that point to the same objects

Defining a new variable does *not* create a new actual object - it only creates another reference to the same object. An example of this error would be:

```
function updateNestedState(state, action) {
  let nestedState = state.nestedState;
  // ERROR: this directly modifies the existing object reference - don't do this!
  nestedState.nestedField = action.data;

  return {
    ...state,
    nestedState
  };
}
```

This function does correctly return a shallow copy of the top-level state object, but because the `nestedState` variable was still pointing at the existing object, the state was directly mutated.

1.2.1.1.2 Common Mistake #2: Only making a shallow copy of one level

Another common version of this error looks like this:

```
function updateNestedState(state, action) {
  // Problem: this only does a shallow copy!
  let newState = {...state};

  // ERROR: nestedState is still the same object!
  newState.nestedState.nestedField = action.data;
```

```

        return newState;
    }
}

```

Doing a shallow copy of the top level is *not* sufficient - the `nestedState` object should be copied as well.

1.2.1.1.3 Correct Approach: Copying All Levels of Nested Data

Unfortunately, the process of correctly applying immutable updates to deeply nested state can easily become verbose and hard to read. Here's what an example of updating `state.first.second[someId].fourth` might look like:

```

function updateVeryNestedField(state, action) {
    return {
        ...state,
        first : {
            ...state.first,
            second : {
                ...state.first.second,
                [action.someId] : {
                    ...state.first.second[action.someId],
                    fourth : action.someValue
                }
            }
        }
    }
}

```

Obviously, each layer of nesting makes this harder to read, and gives more chances to make mistakes. This is one of several reasons why you are encouraged to keep your state flattened, and compose reducers as much as possible.

1.3 INSERTING AND REMOVING ITEMS IN ARRAYS

Normally, a Javascript array's contents are modified using mutative functions like `push`, `unshift`, and `splice`. Since we don't want to mutate state directly in reducers, those should normally be avoided. Because of that, you might see "insert" or "remove" behavior written like this:

```

function insertItem(array, action) {
    return [
        ...array.slice(0, action.index),
        action.item,
        ...array.slice(action.index)
    ]
}

function removeItem(array, action) {
    return [
        ...array.slice(0, action.index),
        ...array.slice(action.index + 1)
    ];
}

```

However, remember that the key is that the *original in-memory reference* is not modified. **As long as we make a copy first, we can safely mutate the copy.** Note that this is true for both arrays and objects, but nested values still must be updated using the same rules.

This means that we could also write the insert and remove functions like this:

```
function insertItem(array, action) {
  let newArray = array.slice();
  newArray.splice(action.index, 0, action.item);
  return newArray;
}

function removeItem(array, action) {
  let newArray = array.slice();
  newArray.splice(action.index, 1);
  return newArray;
}
```

The remove function could also be implemented as:

```
function removeItem(array, action) {
  return array.filter( (item, index) => index !== action.index);
}
```

1.4 UPDATING AN ITEM IN AN ARRAY

Updating one item in an array can be accomplished by using `Array.map`, returning a new value for the item we want to update, and returning the existing values for all other items:

```
function updateObjectInArray(array, action) {
  return array.map( (item, index) => {
    if(index !== action.index) {
      // This isn't the item we care about - keep it as-is
      return item;
    }

    // Otherwise, this is the one we want - return an updated value
    return {
      ...item,
      ...action.item
    };
  });
}
```

1.5 IMMUTABLE UPDATE UTILITY LIBRARIES

Because writing immutable update code can become tedious, there are a number of utility libraries that try to abstract out the process. These libraries vary in APIs and usage, but all try to provide a shorter and more succinct way of writing these updates. Some, like [dot-prop-immutable](#), take string paths for commands:

```
state = dotProp.set(state, `todos.${index}.complete`, true)
```

Others, like [immutability-helper](#) (a fork of the now-deprecated React Immutability Helpers addon), use nested values and helper functions:

```
var collection = [1, 2, {a: [12, 17, 15]}];
var newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});
```

They can provide a useful alternative to writing manual immutable update logic.

[Immutable Data#Immutable Update Utilities](#) section of the [Redux Addons Catalog](#).

17. Outsourcing Action Types

There is a risk of adding a tiny typo in type when dispatching actions to props. Therefore it is good practice to outsource your action types. Use constants to eliminate the danger of mistyping.

Create actions.js in Store folder:

actions.js

```
export const INCREMENT = 'INCREMENT';
export const DECREMENT = 'DECREMENT';
export const ADD = 'ADD';
export const SUBTRACT = 'SUBTRACT';
export const STORE_RESULT = 'STORE_RESULT';
export const DELETE_RESULT = 'DELETE_RESULT';
```

reducer.js

```
import * as actionTypes from './actions';

const initialState = {
  counter: 0,
  results: []
};

const reducer = (state = initialState, action) => {
  if (action.type === actionTypes.INCREMENT) {
    return {
      ...state,
      counter: state.counter + action.val
    };
  }
  if (action.type === actionTypes.STORE_RESULT) {
```

```

        return {
          ...state,
          results: state.results.concat({ id: new Date(), value:
state.counter })
        };
      }
      if (action.type === actionTypes.DELETE_RESULT) {
        const updatedArray = state.results.filter(result => result.id !==
action.resultElId);
        return {
          ...state,
          results: updatedArray
        };
      }

      return state;
   };

export default reducer;

```

Counter.js

```

import React, { Component } from "react";
import { connect } from "react-redux";
import CounterControl from
"../../components/CounterControl/CounterControl";
import CounterOutput from
"../../components/CounterOutput/CounterOutput";
import * as actionTypes from "../../store/actions";

class Counter extends Component {
  state = {
    counter: 0
  };

  counterChangedHandler = (action, value) => {
    switch (action) {
      case "inc":

```

```

    this.setState(prevState => {
      return { counter: prevState.counter + 1 };
    });
    break;
  case "dec":
    this.setState(prevState => {
      return { counter: prevState.counter - 1 };
    });
    break;
  case "add":
    this.setState(prevState => {
      return { counter: prevState.counter + value };
    });
    break;
  case "sub":
    this.setState(prevState => {
      return { counter: prevState.counter - value };
    });
    break;
  }
};

render() {
  return (
    <div>
      {/* <CounterOutput value={this.state.counter} /> */}
      {/* Change the above statement to the below to get counter
from global state*/}
      <CounterOutput value={this.props.ctr} />
      <CounterControl
        label="Increment"
        // clicked={() => this.counterChangedHandler("inc")}
        clicked={this.props.onIncrementCounter}
      />
      <CounterControl
        label="Decrement"
        clicked={() => this.counterChangedHandler("dec")}
      />
      <CounterControl

```

```

        label="Add 5"
        clicked={() => this.counterChangedHandler("add", 5)}
    />
    <CounterControl
        label="Subtract 5"
        clicked={() => this.counterChangedHandler("sub", 5)}
    />

    <hr />
    <button onClick={() =>
this.props.onStoreResult(this.props.ctr)}>
        Store Result
    </button>
    <ul>
        {this.props.storedResults.map(strResult => (
            <li
                key={strResult.id}
                onClick={() => this.props.onDeleteResult(strResult.id)}
            >
                {strResult.value}
            </li>
        ))}
    </ul>
    </div>
);
}
}
// get state and map it to props
const mapStateToProps = state => {
    return {
        ctr: state.counter,
        storedResults: state.results
    };
};

const mapDispatchToProps = dispatch => {
    return {
        // Any no of parameters can be passed. Only the 'type' is
mandatory
    }
};

```

```

    onIncrementCounter: () => dispatch({ type: actionTypes.INCREMENT,
val: 10 }),
    onStoreResult: result => dispatch({ type:
actionTypes.STORE_RESULT, result: result }),
    onDeleteResult: id => dispatch({ type: actionTypes.DELETE_RESULT,
resultElId: id })
};

};

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter);

```

18. Combining Multiple Reducers

All actions at the end get bundles to one reducer.

Redux package gives us a utility method to combine all the reducers into one.

Lets create 2 reducers, one to manage counter and one for results.

To combine reducers, ‘combineReducers’ is used. This function takes a javascript object as input

With reducers combined, redux adds one level of nesting where it has one state object but with ctr and res keys (in this case). So for ex: state.counter becomes state.ctr.counter. state is a global state here.

Note: reducers have access to only their local state.

Counter.js

```

import React, { Component } from "react";
import { connect } from "react-redux";
import CounterControl from
"../../components/CounterControl/CounterControl";
import CounterOutput from
"../../components/CounterOutput/CounterOutput";
import * as actionTypes from "../../store/actions";

class Counter extends Component {
  state = {

```

```

        counter: 0
    };

counterChangedHandler = (action, value) => {
    switch (action) {
        case "inc":
            this.setState(prevState => {
                return { counter: prevState.counter + 1 };
            });
            break;
        case "dec":
            this.setState(prevState => {
                return { counter: prevState.counter - 1 };
            });
            break;
        case "add":
            this.setState(prevState => {
                return { counter: prevState.counter + value };
            });
            break;
        case "sub":
            this.setState(prevState => {
                return { counter: prevState.counter - value };
            });
            break;
    }
};

render() {
    return (
        <div>
            {/* <CounterOutput value={this.state.counter} /> */}
            {/* Change the above statement to the below to get counter
from global state*/}
            <CounterOutput value={this.props.ctr} />
            <CounterControl
                label="Increment"
                //    clicked={() => this.counterChangedHandler("inc")}
                clicked={this.props.onIncrementCounter}

```

```

    />
    <CounterControl
      label="Decrement"
      clicked={() => this.counterChangedHandler("dec")}
    />
    <CounterControl
      label="Add 5"
      clicked={() => this.counterChangedHandler("add", 5)}
    />
    <CounterControl
      label="Subtract 5"
      clicked={() => this.counterChangedHandler("sub", 5)}
    />

    <hr />
    <button onClick={() =>
this.props.onStoreResult(this.props.ctr)}>
      Store Result
    </button>
    <ul>
      {this.props.storedResults.map(strResult => (
        <li
          key={strResult.id}
          onClick={() => this.props.onDeleteResult(strResult.id)}
        >
          {strResult.value}
        </li>
      ))}
    </ul>
  </div>
);
}
}

// get state and map it to props
const mapStateToProps = state => {
  return {
    ctr: state.ctr.counter,
    storedResults: state.res.results
  };
}

```

```

};

const mapDispatchToProps = dispatch => {
  return {
    // Any no of parameters can be passed. Only the 'type' is
    // mandatory
    onIncrementCounter: () => dispatch({ type: actionTypes.INCREMENT,
    val: 10 }),
    onStoreResult: result => dispatch({ type:
    actionTypes.STORE_RESULT, result: result }),
    onDeleteResult: id => dispatch({ type: actionTypes.DELETE_RESULT,
    resultElId: id })
  };
};

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter);

```

index.js

```

import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
import * as serviceWorker from "./serviceWorker";
import { createStore, combineReducers } from "redux";
// Now we have 2 reducers
// import reducer from "./store/reducer";
import counterReducer from "./store/reducers/counter";
import resultReducer from "./store/reducers/result";
import { Provider } from "react-redux";

// This takes a reducer as an input.
/*
  We will have more complex reducers in our application with lot of
  code for different

```

```

types of actions, so we typically store their logics into their own
files

*/
/* We are telling redux that we got 2 feature areas in our
application res and ctr, Use the
reducers for each of them. Merge all of them into obe store, state
and reducer.
*/
const rootReducer = combineReducers({
  ctr: counterReducer,
  res: resultReducer
})
//Pass rootReducer to store
const store = createStore(rootReducer);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("root")
);

// If you want your app to work offline and load faster, you can
// change
// unregister() to register() below. Note this comes with some
// pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();

```

result.js

```

import * as actionTypes from '../actions';

const initialState = {
  results: []
};

```

```

const reducer = ( state = initialState, action ) => {
  switch ( action.type ) {
    case actionTypes.STORE_RESULT:
      return {
        ...state,
        results: state.results.concat({id: new Date(), value: action.result})
      }
    case actionTypes.DELETE_RESULT:
      // const id = 2;
      // const newArray = [...state.results];
      // newArray.splice(id, 1)
      const updatedArray = state.results.filter(result => result.id !== action.resultElId);
      return {
        ...state,
        results: updatedArray
      }
  }
  return state;
};

export default reducer;

```

counter.js

```

import * as actionTypes from '../actions';

const initialState = {
  counter: 0
};

const reducer = ( state = initialState, action ) => {
  switch ( action.type ) {
    case actionTypes.INCREMENT:
      const newState = Object.assign({}, state);
      newState.counter = state.counter + 1;
      return newState;
    case actionTypes.DECREMENT:
  }
}

```

```
        return {
          ...state,
          counter: state.counter - 1
        }
      case actionTypes.ADD:
        return {
          ...state,
          counter: state.counter + action.val
        }
      case actionTypes.SUBTRACT:
        return {
          ...state,
          counter: state.counter - action.val
        }
      }
      return state;
};

export default reducer;
```

19. Understanding State Types

Should every state be handles through redux?

[19.1 state-types.pdf](#)

Type	Example	Use Redux?
Local UI State	Show / Hide Backdrop	Mostly handled within components
Persistent State	All Users, all Posts, ...	Stored on Server, relevant slice managed by Redux
Client State	Is Authenticated? Filters set by User, ...	Managed via Redux

20. Time to Practice - Redux Basics

21. Combining Local UI State and Redux

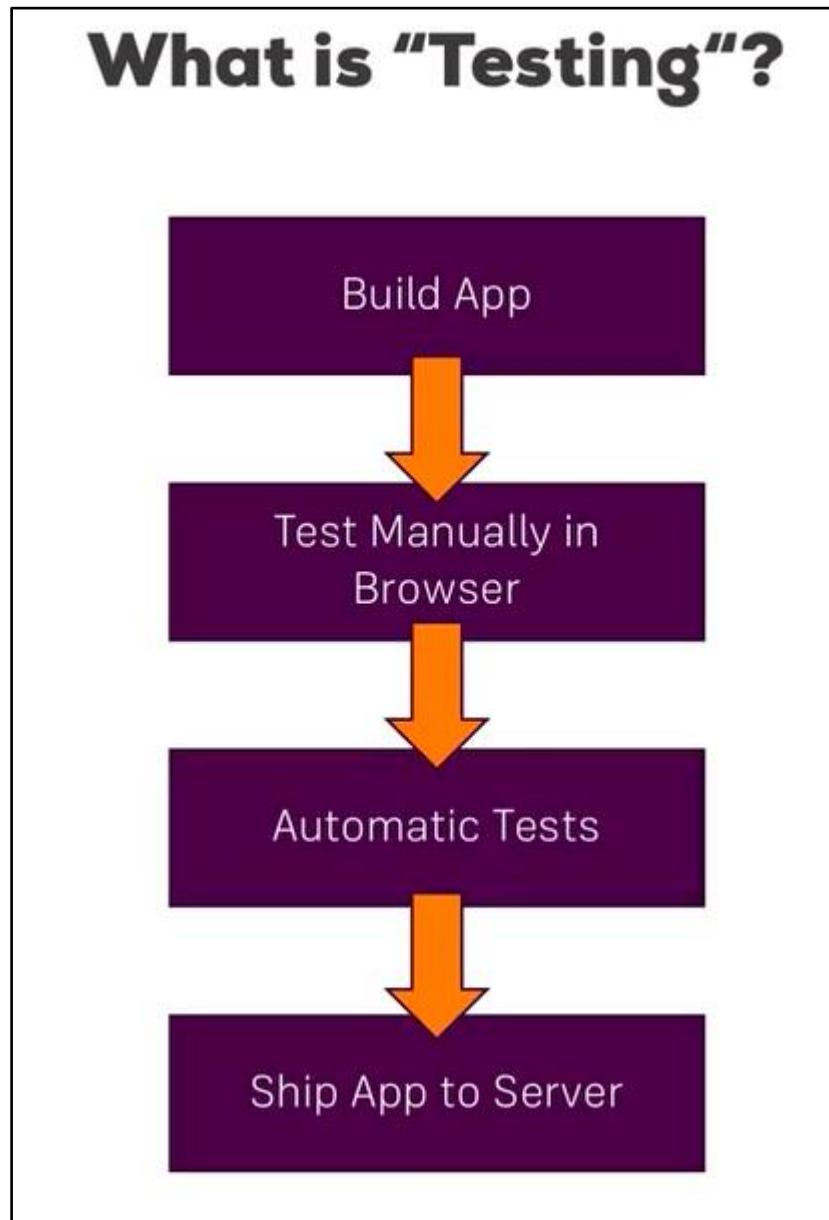
22. Wrap Up

23. Useful Resources _ Links

2. Redux Docs: <http://redux.js.org/>
3. Core Concepts: <http://redux.js.org/docs/introduction/CoreConcepts.html>
4. Actions: <http://redux.js.org/docs/basics/Actions.html>
5. Reducers: <http://redux.js.org/docs/basics/Reducers.html>
6. Redux FAQs: <http://redux.js.org/docs/FAQ.html>

20. Testing

1. Module Introduction
2. What is Testing



Why Testing?

	First Draft	New Feature
Component should output "Hello"	PASS	PASS
Component should always render component "PersonData"	PASS	PASS
Component should always receive a "persons" prop	PASS	FAIL
Component should always render the "NewPerson" component when a "editable" (true) prop is received	PASS	FAIL

3. Required Testing Tools

Testing Tools

Test Runner

Executes Tests and provides Validation Library

Jest

Testing Utilities

"Simulates" the React App (mounts components, allows you to dig into the DOM)

React Test Utils

Enzyme

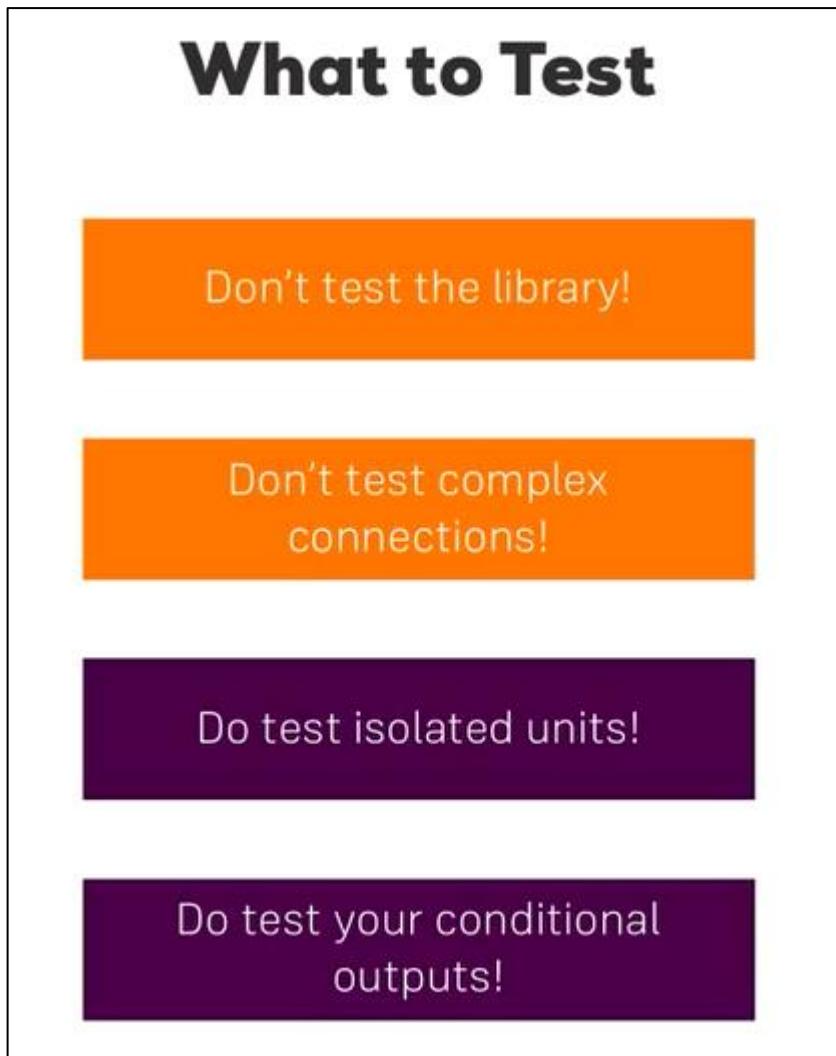
We need 2 tools to be able to write unit testing:

- 1) Test Runner → Runs the test code. Provides a validation library which at the end allows us to do comparisons and potentially throw some errors. The idea behind unit test is that they do not run in the browser but instead with node.js though often emulated to be in a browser environment with the help of specific javascript packages. Create-react-app already comes up with the pre-testing testing environment we can build up on.

- 2) JEST → Jest is already installed in the app created with create-react-app. Javascript testing tool not limited to react but often used in react apps.

Note: Running the test is one thing, when working with react and components we also need a way of emulating these components, basically mounting them to some non-existent DOM and then traversing our components. And we want to do it in an easy way without having to create the whole component tree which might also introduce some side effects. For this we need Testing utilities. ‘React Test Utils’ is the official utility tool. There is one more tool suggested by react team, Enzyme. Enzyme is a tool developed by Airbnb, and it makes it easy to mount components and navigate through them.

4. 4.What To Test



5. 5.Writing our First Test

JEST is already installed.

npm install –save enzyme react-test-renderer enzyme-adapter-react-16 → react-test-renderer help enzyme to work with react and jest correctly. enzyme-adapter-react-16 is the adaptor of the enzyme package to the current react version.

6. 6.Testing Components Continued

<filename>.test.js is automatically picked up by create-react-app

Testing NavigationItems.js

Most of your react components are just functions therefore they only depend on the props they receive, this is something you have to keep in mind for testing.

Create a file NavigationItems.test.js in NavigationItems folder. .test.js files are automatically icked up by react once we run “”.

7.

8. 7.Jest and Enzyme Documentations

9.

10.8.Testing Components Correctly

11.

12.9.Testing Containers

13.

14.10.How to Test Redux

15.

16.11.Wrap Up

17.

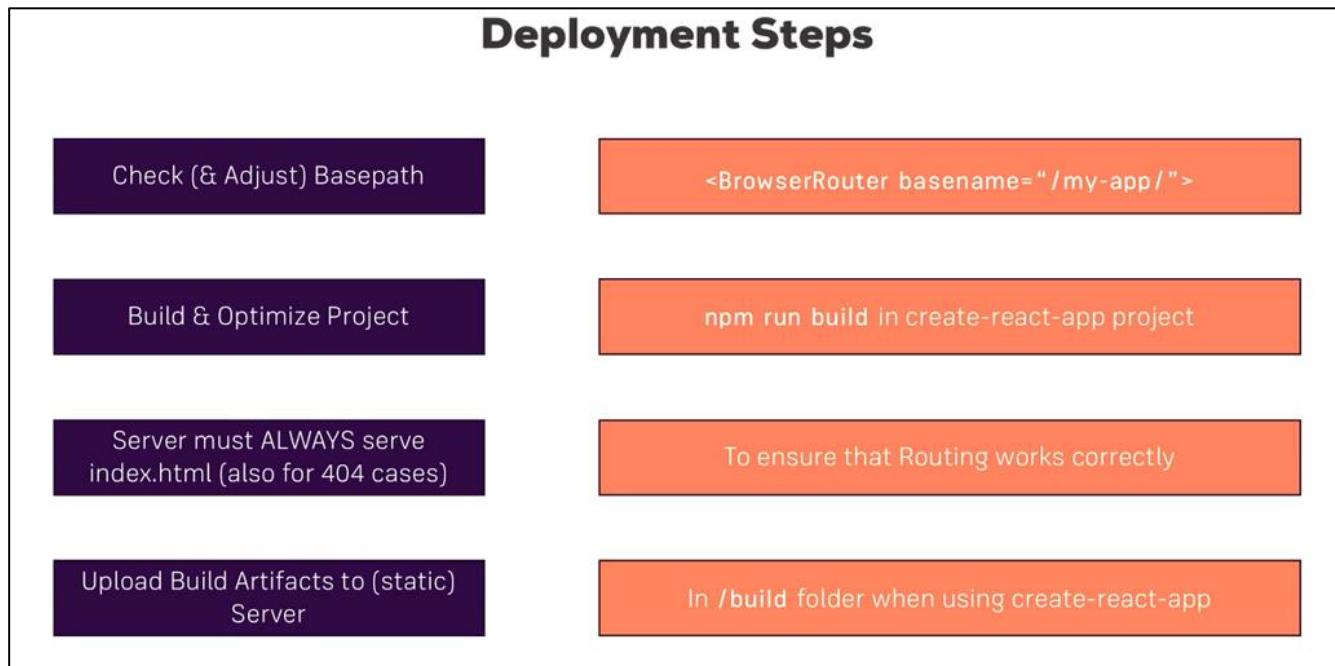
18.12. Useful Resources & Links

- Enzyme API: <http://airbnb.io/enzyme/docs/api/>
- Jest Docs: <https://facebook.github.io/jest/>

21. Deploying the App to the Web

1. Module Introduction

7. Deployment Steps



8. Building the Project

4. Example Deploying on Firebase

5. Wrap Up

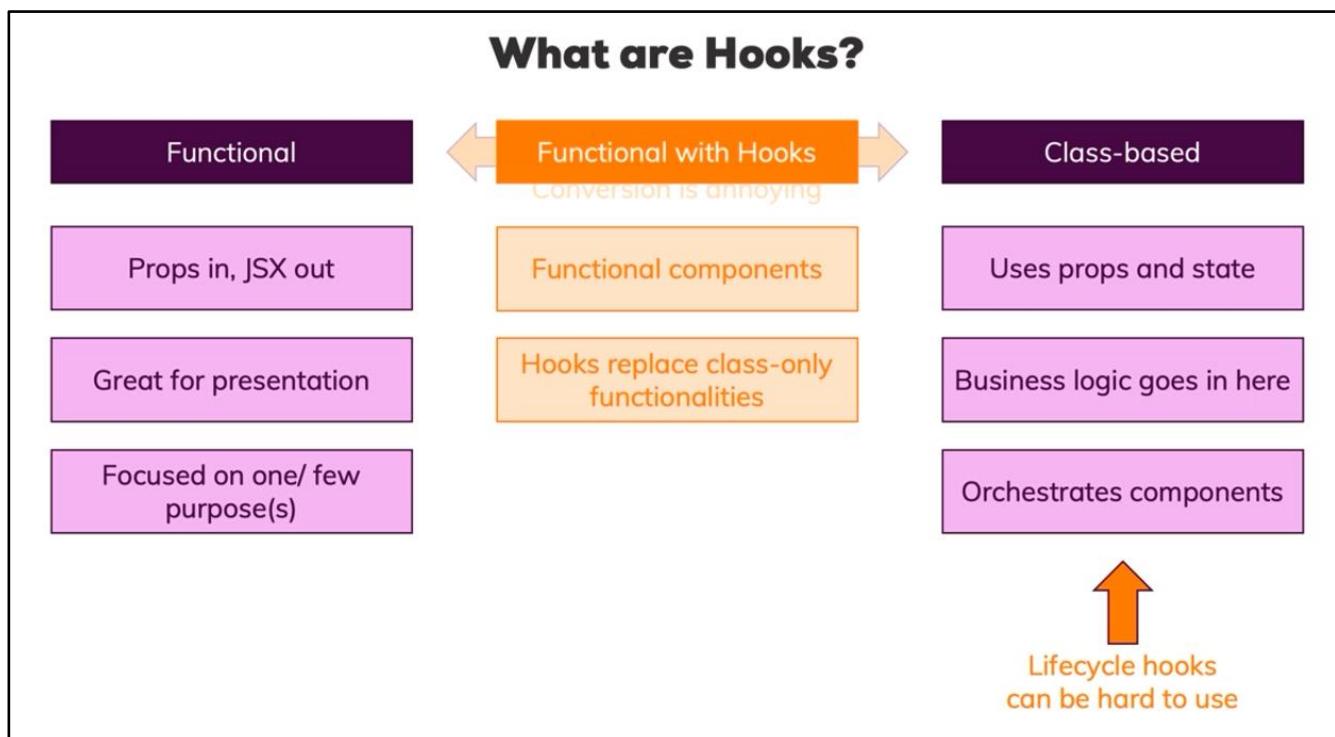
26. React Hooks

1. Introduction

Added to react in react 16.8.

- Allows you to use functional component only.
- Even state can be managed in functional components using hooks.

2. What are Hooks



Problems with old approach:

- 1) Conversion from functional to class or vice-versa is annoying.
- 2) Lifecycle hooks can be hard to use in class based components. Knowing when to use what lifecycle and how to use it correctly can be challenging specially in more complex application.

→ Hooks are extra feature/function we can call in our functional component which gives us access to certain capability we could only use in class based components before.

3. Enabling Hooks

Make sure your react version is at least 16.8.

Delete node modules → Browse to project folder → run 'npm install'

4. The useState() Hook

Capture user input in Todo.js. Historically to capture user input we needed class based component with **2-way binding**, where we can declare some event handler to update the state and then can use the same state to populate the value.

But with react 16.8 onwards, we can achieve this with hooks.

```
import React, { useState } from "react";
```

'use' at the start of the 'useState' signals that this is a hook function. It allows us to hook into a certain react functionality.

How to capture user input? → useState()

useState() takes an initial state. Lets say we want to manage user input and initially it should be empty so we can pass an empty string like, useState(""). However we can pass anything in useState(), an empty string, empty array, empty object, 0, null. **It returns a value which is an array and this array will have exactly 2 elements where the first element is the current state and the 2nd element is the function which we can use to manipulate the state.**

```
onChange={inputState[1]}
```

will return the complete event but we want value so write a function.

The screenshot shows a code editor with a file named 'Todo.js'. The code uses the useState hook to manage a todo list. An input field is present with an onChange prop set to a function that updates the state. The browser window shows the text 'Hello react!1'. A red arrow points from the browser's input field to the code editor's onChange prop, illustrating how the prop returns an event object.

```
JS Todo.js x
1 import React, { useState } from "react";
2 const todo = props => {
3   const [inputState, setInputState] = useState("");
4   const [inputChangeHandler] = (event)=> {
5     setInputState([event.target.value]);
6   }
7
8   return (
9     <React.Fragment>
10    <input type="text" placeholder="Todo" onChange={inputChangeHandler} value={inputState}>
11    <button type="button">Add</button>
12    <ul />
13  </React.Fragment>
14 );
15 };
16 export default todo;
17
```

Todo.js

```
import React, { useState } from "react";
const todo = props => {
  const [inputState, setInputState] = useState("");
  const [inputChangeHandler] = (event)=> {
    setInputState([event.target.value]);
  }

  return (
```

```

<React.Fragment>
  <input type="text" placeholder="Todo"
  onChange={inputChangeHandler} value={inputState[0]} />
  <button type="button">Add</button>
  <ul />
</React.Fragment>
);
};

export default todo;

```

O/P:

The screenshot shows a code editor with a file named `Todo.js`. The code uses the `useState` hook from React to manage state. It includes a function component that returns a `React.Fragment` containing an `input` field and a `button`. The browser preview shows a simple UI with an input field containing "Test" and a button labeled "Add". The URL is `https://o000h.codesandbox.io/`.

```

JS Todo.js  x
1 import React, { useState } from "react";
2 const todo = props => {
3   const inputState = useState("");
4   const inputChangeHandler = (event)=> {
5     inputState[1](event.target.value);
6   }
7
8   return (
9     <React.Fragment>
10    <input type="text" placeholder="Todo"
11      onChange={inputChangeHandler}
12      value={inputState[0]} />
13    <button type="button">Add</button>
14    <ul />
15  </React.Fragment>
16 );
17 };
18 export default todo;
19

```

5. Adding Array Destructuring

Example of array destructuring:

```

>>> const arr = ["value", function fun(){console.log("I am a function")}];
undefined

>>const [val, displayFn] = arr;
undefined

>>val
"value"

>>displayFn
f fun(){console.log("I am a function")}

```

Todo.js

```
import React, { useState } from "react";
const todo = props => {
  const [todoName, setTodoName] = useState('');
  const inputChangeHandler = (event)=> {
    setTodoName(event.target.value);
  }

  return (
    <React.Fragment>
      <input type="text" placeholder="Todo"
        onChange={inputChangeHandler}
        value={todoName}/>
      <button type="button">Add</button>
      <ul />
    </React.Fragment>
  );
};

export default todo;
```

6. Using Multiple States

Output Todo list.

Todo.js

```
import React, { useState } from "react";
const todo = props => {
  const [todoName, setTodoName] = useState('');
  const [todoList, setTodoList] = useState([]);
  const inputChangeHandler = (event)=> {
    setTodoName(event.target.value);
  }
  const todoAddHandler = () => {
    // concat() will return a new array
    setTodoList(todoList.concat(todoName))
  }

  return (
    <React.Fragment>
      <input type="text" placeholder="Todo"
```

```

        onChange={inputChangeHandler}
        value={todoName}/>
      <button type="button"
onClick={todoAddHandler}>Add</button>
    <ul>
      {todoList.map(todo=> <li key={todo}>{todo}</li>) }
    </ul>
  </React.Fragment>
);
};

export default todo;

```

O/P:



7. Using One State Instead

However it is recommended to use separate hooks to manage each individual state. Reasons are below:

- ➔ Unlike `useState()`, hooks will not be able to merge your current state with the previous state rather it will just replace the old state with the new one.

Todo.js

```

import React, { useState } from "react";
const todo = props => {
  // const [todoName, setTodoName] = useState('');
  // const [todoList, setTodoList] = useState([]);
  const [todoState, setTodoState] = useState({ userInput: '',
todoList: [] });

  const inputChangeHandler = event => {
    setTodoState({

```

```

    userInput: event.target.value,
    todoList: todoState.todoList
  ) );
};

const todoAddHandler = () => {
  setTodoState({
    userInput: todoState.userInput,
    todoList:
  todoState.todoList.concat(todoState.userInput)
  ) );
};

return (
  <React.Fragment>
    <input
      type="text"
      placeholder="Todo"
      onChange={inputChangeHandler}
      value={todoState.userInput}
    />
    <button type="button" onClick={todoAddHandler}>
      Add
    </button>
    <ul>
      {todoState.todoList.map(todo => (
        <li key={todo}>{todo}</li>
      )));
    </ul>
  </React.Fragment>
);
};

export default todo;

```

8. The Rules of Hooks

- ➔ You must only use all the hooks at the top level of your component function (A function that takes props and returns JSX).

➔ You must always call useState() at the top level of your function body. You cannot call useState() at following places:

- a) Inside a function
- b) In if block
- c) Any other form of nesting

9. Sending Data via Http

Let's say, we want to save list on the server. Just to test setup a realtime database in google firebase. Take the url exposed by firebase and make rest calls. <https://fir-6cd72.firebaseio.com/>

Add todo.json at the end of the url as this becomes the node under which we want to save our data.

Todo.js

```
import React, { useState } from "react";
import axios from "axios";

const todo = props => {
  const [todoName, setTodoName] = useState('');
  const [todoList, setTodoList] = useState([]);
  const inputChangeHandler = (event)=> {
    setTodoName(event.target.value);
  }
  const todoAddHandler = () => {
    // concat() will return a new array
    setTodoList(todoList.concat(todoName));
    axios.post('https://fir-
6cd72.firebaseio.com/todo.json', {name:todoName})
      .then(res => {
        console.log(res);
      }).catch(err => {
        console.log(err);
      })
  }

  return (
    <React.Fragment>
      <input type="text" placeholder="Todo"
```

```

        onChange={inputChangeHandler}
        value={todoName} />
      <button type="button"
onClick={todoAddHandler}>Add</button>
      <ul>
        {todoList.map(todo=> <li>{todo}</li>) }
      </ul>
    </React.Fragment>
  );
};

export default todo;

```

Firebase screen:



10. The useEffect() Hook

How to fetch data? Fetch the data when the component gets loaded. Use useEffect() hook.

You pass a function to useEffect() that should be executed when the component loads for the first time (that is our case).

Todo.js

```

import React, { useState, useEffect } from "react";
import axios from "axios";

const todo = props => {
  const [todoName, setTodoName] = useState('');

```

```

const [todoList, setTodoList] = useState([]);

// Pass a function to useEffect()
useEffect(() => {
  axios.get('https://fir-
6cd72.firebaseio.com/todo.json').then(result => {
    // console.log(result);
    const todoData = result.data;
    const todos = [];
    for (const key in todoData) {
      todos.push({id: key, name: todoData[key].name})
    }
    // This changes state and we enter into infinite loop
    setTodoList(todos);
  }) ;
}) ;

const inputChangeHandler = (event)=> {
  setTodoName(event.target.value);
}

const todoAddHandler = () => {
  // concat() will return a new array
  setTodoList(todoList.concat(todoName));
  axios.post('https://fir-
6cd72.firebaseio.com/todo.json', {name:todoName})
  .then(res => {
    console.log(res);
  }).catch(err => {
    console.log(err);
  })
}

return (
<React.Fragment>
  <input type="text" placeholder="Todo"
  onChange={inputChangeHandler}
  value={todoName}/>

```

```

        <button type="button"
onClick={todoAddHandler}>Add</button>
        <ul>
            {todoList.map(todo=> <li
key={todo.id}>{todo.name}</li>) }
        </ul>
    </React.Fragment>
);
};

export default todo;

```

O/P:



Note: How do we fix the infinite loop issue here?

11. Controlling Effect Execution

useEffect() runs after every cycle unlike the componentDidMount() that ran only once.

useEffect() takes 2 arguments. It does not just take the 1st argument which is the function it executes but also takes a 2nd argument which is an array of values we want to have a look at before it executes the function (1st argument). Only if the values specified in the array changes this effect should run again.

So, different options for 2nd arguments:

- 1) [] → Equivalent to componentDidMount(). As react has nothing to watch therefore it will not detect any change and first argument function will never run again.
- 2) [someValue] → Equivalent to componentDidMount() + componentDidUpdate() with an if check. React will only run the effect when it detects a change in 'someValue'.

Note: The 2nd argument can be used to stop infinite loop.

12. Effect Cleanup

Q) How to get componentDidUnmount() behavior? What if you have side effects that require clean up work.

Ans: Use return statement. This return statement should also be a function. This function will be executed by react on every render cycle. React will execute this as a cleanup before it applies the effect of your main code again.

Todo.js (snippet)

```
useEffect(() => {
  axios.get('https://fir-
6cd72.firebaseio.com/todo.json').then(result => {
    // console.log(result);
    const todoData = result.data;
    const todos = [];
    for (const key in todoData) {
      todos.push({id: key, name: todoData[key].name})
    }
    setTodoList(todos);
  });
  return ()=>{
    console.log("cleanup");
  }
}, [todoName]);
```

We can have multiple effects in our component. Let's say we have an event listener. We want to listen the event whenever the user moves mouse.

Todo.js

```
useEffect(()=>{
  document.addEventListener('mousemove', event=>{
    console.log(event.clientX, event.clientY);
  })
});
```

O/P:

We can see that as we type in textbox, more and more event listeners get added which is a serious performance issue.

40 115 182	common-sandbox.52624a55.chunk.js:15
40 107 200	common-sandbox.52624a55.chunk.js:15
40 88 230	common-sandbox.52624a55.chunk.js:15
40 75 245	common-sandbox.52624a55.chunk.js:15
40 45 278	common-sandbox.52624a55.chunk.js:15
40 18 300	common-sandbox.52624a55.chunk.js:15
40 9 358	common-sandbox.52624a55.chunk.js:15
40 9 361	common-sandbox.52624a55.chunk.js:15
40 12 365	common-sandbox.52624a55.chunk.js:15
40 17 375	common-sandbox.52624a55.chunk.js:15

So the solution is that we need to clean all the listener before we attach a new one.

Todo.js

```
const mouseMoveHandler = event => {
    console.log(event.clientX, event.clientY);
}

useEffect(()=>{
    document.addEventListener('mousemove', mouseMoveHandler);
    // return is used to write clean up code
    return(()=>{
        // Remove the mousemove listener that executes
        mouseMoveHandler
        document.removeEventListener('mousemove',
        mouseMoveHandler);
    });
}) ;
```

Now, what if we do not want to clean up for every render cycle (here for every key stroke), rather clean up only when the component gets destroyed. Add an empty array as 2nd argument.

Todo.js

```
import React, { useState, useEffect } from "react";
import axios from "axios";

const todo = props => {
```

```

const [todoName, setTodoName] = useState('');
const [todoList, setTodoList] = useState([]);

// Pass a function to useEffect() as its 1st argument
useEffect(() => {
    axios.get('https://fir-
6cd72.firebaseio.com/todo.json').then(result => {
        // console.log(result);
        const todoData = result.data;
        const todos = [];
        for (const key in todoData) {
            todos.push({id: key, name: todoData[key].name})
        }
        setTodoList(todos);
    });
    return ()=>{
        console.log("cleanup");
    }
}, [todoName]);

const mouseMoveHandler = event => {
    console.log(event.clientX, event.clientY);
}

// Only run on mounting and unmounting now
useEffect(()=>{
    document.addEventListener('mousemove', mouseMoveHandler
);
    // return is used to write clean up code
    return(()=>{
        // Remove the mousemove listener that executes
        mouseMoveHandler
        document.removeEventListener('mousemove',
        mouseMoveHandler);
    });
}, []);

const inputChangeHandler = (event)=> {

```

```

        setTodoName(event.target.value);
    }
    const todoAddHandler = () => {
        // concat() will return a new array
        setTodoList(todoList.concat(todoName));
        axios.post('https://fir-
6cd72.firebaseio.com/todo.json', {name:todoName})
            .then(res => {
                console.log(res);
            }).catch(err => {
                console.log(err);
            })
    }

    return (
        <React.Fragment>
            <input type="text" placeholder="Todo"
                onChange={inputChangeHandler}
                value={todoName}/>
            <button type="button"
                onClick={todoAddHandler}>Add</button>
            <ul>
                {todoList.map(todo=> <li
                    key={todo.id}>{todo.name}</li>)}
            </ul>
        </React.Fragment>
    );
};

export default todo;

```

O/P:

```

102 93 common-sandbox.52624a55.chunk.js:15
101 92 common-sandbox.52624a55.chunk.js:15
99 91 common-sandbox.52624a55.chunk.js:15
97 91 common-sandbox.52624a55.chunk.js:15
95 89 common-sandbox.52624a55.chunk.js:15
95 88 common-sandbox.52624a55.chunk.js:15
94 88 common-sandbox.52624a55.chunk.js:15
92 85 common-sandbox.52624a55.chunk.js:15
91 85 common-sandbox.52624a55.chunk.js:15

```

13. Converting the App Component

useState() → to manage state

useEffect() → To manage side effects (http calls, pagewise event listeners)

Add Header.js and Auth.js.

Switch between Todo and Auth component. When clicked on ‘Todo List’ button, display <Todo/> component and Show <Auth/> when clicked on ‘Auth’ button.

Header.js

```

import React from "react";

const header = (props) => {
  return (
    <header>
      <button onClick={props.onLoadTodos}>Todo List</button> { ' }
    <button onClick={props.onLoadAuth}>Auth</button>
    </header>
  );
}

export default header;

```

Auth.js

```

import React from "react";

const Auth = (props) => (
  <h1> Auth Component </h1>
);
export default Auth;

```

Todo.js

```
import React, { useState, useEffect } from "react";
import axios from "axios";

const Todo = props => {
  const [todoName, setTodoName] = useState('');
  const [todoList, setTodoList] = useState([]);

  // Pass a function to useEffect() as its 1st argument
  useEffect(() => {
    axios.get('https://fir-
6cd72.firebaseio.com/todo.json') .then(result => {
      // console.log(result);
      const todoData = result.data;
      const todos = [];
      for (const key in todoData) {
        todos.push({id: key, name: todoData[key].name})
      }
      setTodoList(todos);
    }) ;
    return ()=>{
      console.log("cleanup");
    }
  }, [todoName]);

  const mouseMoveHandler = event => {
    console.log(event.clientX, event.clientY);
  }

  useEffect(()=>{
    document.addEventListener('mousemove', mouseMoveHandler)
  );
  // return is used to write clean up code
  return(()=>{
    // Remove the mousemove listener that executes
    mouseMoveHandler
  })
}
```

```

        document.removeEventListener('mousemove',
mouseMoveHandler);
    } );
}, [] );
}

const inputChangeHandler = (event)=> {
    setTodoName(event.target.value);
}

const todoAddHandler = () => {
    // concat() will return a new array
    setTodoList(todoList.concat(todoName));
    axios.post('https://fir-
6cd72.firebaseio.com/todo.json', {name:todoName})
    .then(res => {
        console.log(res);
    }).catch(err => {
        console.log(err);
    })
}

return (
<React.Fragment>
    <input type="text" placeholder="Todo"
        onChange={inputChangeHandler}
        value={todoName}/>
    <button type="button"
        onClick={todoAddHandler}>Add</button>
    <ul>
        {todoList.map(todo=> <li
key={todo.id}>{todo.name}</li>) }
    </ul>
</React.Fragment>
) ;
}
export default Todo;

```

App.js

```

import React, {useState} from "react";
import Todo from "./components/Todo";
import Header from "./components/Header";
import Auth from "./components/Auth";

const App = (props) => {

  const [page, setPage] = useState('auth');

  const switchPage = (pageName) => {
    setPage(pageName);
  }

  return (
    <div className="App">
      <Header onLoadTodos={switchPage.bind(this,'todos')}>
        onLoadAuth = {switchPage.bind(this,'auth')}
      />
      <hr/>
      {page==='auth'? <Auth/> : <Todo/> }
    </div>
  );
}

export default App;

```

O/P:



14.The useContext() Hook

Fake a login mechanism. Click ‘Log In’ button to fake logon and use context to set status.

Context allows you to pass state/values across components without using props all the time.

Create auth-context.js.

auth-context.js

```
import React from "react";

// false is the default value for the context
// Keeping default value as false as we want to start as
unauthenticated
// The value can be an object, number etc.
// We will use this context in App.js

/* By setting the same context structure here as we have in
App.js,
The IDE gives better autocompletion
*/
const AuthContext = React.createContext({ status: false,
login: () => {} });

export default AuthContext;
```

In order to use **Context** defined in auth-context.js, we need to set up a **Provider** in App.js (as we want to use context in App.js). Wrap everything in App.js with <AuthContext.Provider> that should be able to use the context.

Also create a function ‘login’ in App.js to change the auth status.

App.js

```
import React, { useState } from "react";
import Todo from "./components/Todo";
import Header from "./components/Header";
import Auth from "./components/Auth";
import AuthContext from "./auth-context";

const App = props => {
  const [page, setPage] = useState("auth");
  const [authStatus, setAuthStatus] = useState(false);

  return (
    <AuthContext.Provider value={{ status: authStatus, login: setAuthStatus }}>
      <Header />
      <Todo />
    </AuthContext.Provider>
  );
}

export default App;
```

```

const switchPage = pageName => {
  console.log(pageName);
  setPage(pageName);
};

const login = () => {
  setAuthStatus(true);
};

return (
  <div className="App">
    {/* Overriding Context values using 'value' property */}
    {/* login is reference to login function */}
    <AuthContext.Provider value={{ status: authStatus,
      login: login }}>
      <Header
        onLoadTodos={switchPage.bind(this, "todos")}
        onLoadAuth={switchPage.bind(this, "auth")}>
      />
      <hr />
      {page === "auth" ? <Auth /> : <Todo />}
    </AuthContext.Provider>
  </div>
);
};

export default App;

```

Use the context in Auth.js. To use Context, first make sure that Auth.js is not an inline function which instantly returns JSX. Instead have a function body and use return statement to return JSX.

To access context use **useContext** hook.

Auth.js

```

import React, { useContext } from "react";
import AuthContext from "../auth-context";

const Auth = props => {

```

```

/* useContext is used to get access to the context.
   There can be more than one Context in react,
   so we need an identifier for the context we want to tap
in here.

   AuthContext is that identifier.

*/
const auth = useContext(AuthContext);

return <button onClick={auth.login}>Log in!</button>;
};

export default Auth;

```

Output current auth status just to show that it works. Only unlock the ‘Todo List’ button if we are authenticated. For this we need to get access to context in Header.js.

Header.js

```

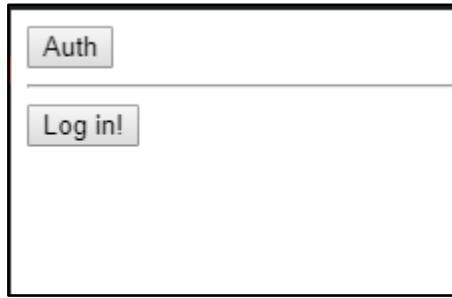
import React, { useContext } from "react";
import AuthContext from "../auth-context";

const header = props => {
  const auth = useContext(AuthContext);
  return (
    <header>
      {auth.status ? (
        <button onClick={props.onLoadTodos}>Todo
List</button>
      ) : null}
      <button onClick={props.onLoadAuth}>Auth</button>
    </header>
  );
};

export default header;

```

O/P:



O/P after clicking 'Log in!' in above screen:



15.State and Effects Gotchas

Note: An issue in the current project is when we input a name in 'Todo List' and click 'Add', the item does not get added to the list below textbox.



Fix in ToDo.js.

```
import React, { useState, useEffect } from "react";
import axios from "axios";

const Todo = props => {
  const [todoName, setTodoName] = useState("");
  const [todoList, setTodoList] = useState([]);

  // Pass a function to useEffect() as its 1st argument
  useEffect(() => {
```

```

    axios.get("https://fir-
6cd72.firebaseio.com/todo.json") .then(result => {
      const todoData = result.data;
      const todos = [];
      for (const key in todoData) {
        todos.push({ id: key, name: todoData[key].name });
      }
      setTodoList(todos);
    }) ;
    return () => {
      console.log("cleanup");
    };
}, [todoName]);

const mouseMoveHandler = event => {
  console.log(event.clientX, event.clientY);
};

useEffect(() => {
  document.addEventListener("mousemove", mouseMoveHandler);
  // return is used to write clean up code
  return () => {
    // Remove the mousemove listener that executes
    mouseMoveHandler
    document.removeEventListener("mousemove",
mouseMoveHandler);
  };
}, []);

const inputChangeHandler = event => {
  setTodoName(event.target.value);
};

const todoAddHandler = () => {
  axios
    .post("https://fir-6cd72.firebaseio.com/todo.json", {
      name: todoName })
    .then(res => {

```

```
        const todoItem = { id: res.data.name, name: todoName
    } ;
        // concat() will return a new array
        setTodoList(todoList.concat(todoItem));
    })
    .catch(err => {
        console.log(err);
    }) ;
}

return (
<React.Fragment>
<input
    type="text"
    placeholder="Todo"
    onChange={inputChangeHandler}
    value={todoName}
/>
<button type="button" onClick={todoAddHandler}>
    Add
</button>
<ul>
    {todoList.map(todo => (
        <li key={todo.id}>{todo.name}</li>
    )));
</ul>
</React.Fragment>
);
};

export default Todo;
```

O/P:



Now we are able to add items, but we can run into an issue in Todo.js. **What if a response from a post request takes longer time?** We have deliberately prolonged a post response using setTimeout().

Todo.js

```
const todoAddHandler = () => {
    axios
        .post("https://fir-6cd72.firebaseio.com/todo.json", {
name: todoName })
        .then(res => {
            setTimeout(() => {
                const todoItem = { id: res.data.name, name:
todoName };
                // concat() will return a new array
                setTodoList(todoList.concat(todoItem));
            }, 5000);
        })
        .catch(err => {
            console.log(err);
        });
};
```

In the Todo List page, lets say we add an entry ‘Test’ and we modify this entry to ‘Test1’ before 3 seconds timeout is elapsed. In this case ‘Test1’ overrides ‘Test’ in the list below the textbox. However if we refresh the page, we can see both the entries fetched from firebase. So we are doing something wrong when we are updating the added value in the front end. The time we click ‘Add’ button, we enter into **todoAddHandler()** which essentially is a closure which means that the variable values that we get from the outside is locked in at the point of time this function starts execution. Therefore our first and second todos that we add, build up on the same starting todoList. In other words, when we

add the 2nd todos before the 1st todos was added to the state, we have the same starting state. So the list does not reflect the first todo added simply bcz the value is locked in the function at the point of time it starts executing. This can be fixed using hooks.

In Todo.js

1) Add state. This extra state is used to trigger the extra render cycle,to store todo and therefore break out of the closure and use the latest state of todoList.

```
const [submittedTodo, setSubmittedTodo] = useState(null);
```

2)Add todos in state

```
const todoAddHandler = () => {
    axios
        .post("https://fir-6cd72.firebaseio.com/todo.json", {
            name: todoName })
        .then(res => {
            setTimeout(() => {
                const todoItem = { id: res.data.name, name:
                    todoName };
                setSubmittedTodo(todoItem);
            }, 3000);
        })
        .catch(err => {
            console.log(err);
        });
};
```

3)Create useEffect that will run on every render cycle (only when the submittedTodo changes) and update todoList. Render cycle will be called when we call setSubmittedTodo(todoItem) as this changes the state.

```
useEffect(() => {
    if (submittedTodo) {
        setTodoList(todoList.concat(submittedTodo));
    }
}, [submittedTodo]);
```

Todo.js

```

import React, { useState, useEffect } from "react";
import axios from "axios";

const Todo = props => {
  const [todoName, setTodoName] = useState("");
  const [submittedTodo, setSubmittedTodo] = useState(null);
  const [todoList, setTodoList] = useState([]);

  // Pass a function to useEffect() as its 1st argument
  useEffect(() => {
    axios.get("https://fir-
6cd72.firebaseio.com/todo.json").then(result => {
      // console.log(result);
      const todoData = result.data;
      const todos = [];
      for (const key in todoData) {
        todos.push({ id: key, name: todoData[key].name });
      }
      setTodoList(todos);
    });
    return () => {
      console.log("cleanup");
    };
  }, [todoName]);

  const mouseMoveHandler = event => {
    console.log(event.clientX, event.clientY);
  };

  useEffect(() => {
    document.addEventListener("mousemove", mouseMoveHandler);
    // return is used to write clean up code
    return () => {
      // Remove the mousemove listener that executes
      mouseMoveHandler
      document.removeEventListener("mousemove",
      mouseMoveHandler);
    };
  });
}

```

```

    } ;
}, [ ]) ;

useEffect(() => {
  if (submittedTodo) {
    setTodoList(todoList.concat(submittedTodo));
  }
}, [submittedTodo]);

const inputChangeHandler = event => {
  setTodoName(event.target.value);
};

const todoAddHandler = () => {
  axios
    .post("https://fir-6cd72.firebaseio.com/todo.json", {
      name: todoName
    })
    .then(res => {
      setTimeout(() => {
        const todoItem = { id: res.data.name, name: todoName };
        setSubmittedTodo(todoItem);
      }, 3000);
    })
    .catch(err => {
      console.log(err);
    });
};

return (
  <React.Fragment>
    <input
      type="text"
      placeholder="Todo"
      onChange={inputChangeHandler}
      value={todoName}
    />
    <button type="button" onClick={todoAddHandler}>

```

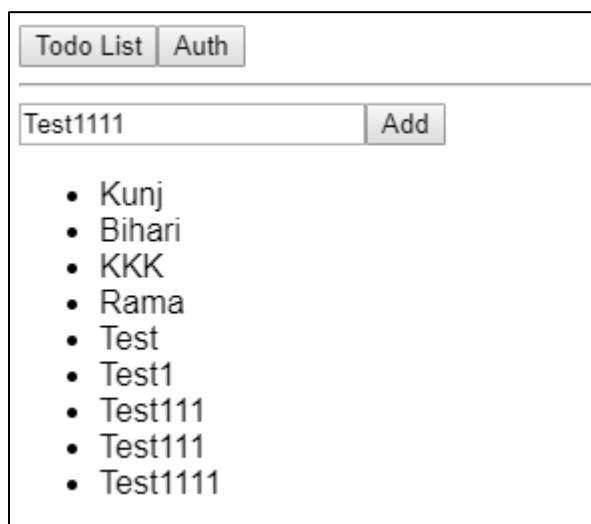
```

        Add
    </button>
    <ul>
        {todoList.map(todo => (
            <li key={todo.id}>{todo.name}</li>
        )));
    </ul>
    </React.Fragment>
);
};

export default Todo;

```

O/P:



16.The useReducer() Hook

Click an item and get rid of it (next lecture). This can be achieved using click handler where the handler can send a delete request but here we are using useReducer hook. **A reducer is a function which can handle couple of different conditions and update state in different ways. useReducer() allows us to bundle the logic for updating the state in one simple function.**

useReducer is independent of redux. It is built in core react. It is a simple function that helps us to manipulate state conveniently.

Todo.js

```

import React, { useState, useEffect, useReducer } from
"react";
import axios from "axios";

const Todo = props => {
  const [todoName, setTodoName] = useState("");
  const [submittedTodo, setSubmittedTodo] = useState(null);
  // As we are using useReducer. Comment this line of code
  //const [todoList, setTodoList] = useState([]);

  //React will pass these arguments automatically for us
  // state -> latest state
  // action -> Info about what to do
  const todoListReducer = (state, action) => {
    switch (action.type) {
      case "ADD":
        return state.concat(action.payload);
        // this will set my state to a completely new list
      case "SET":
        return action.payload;
      case "REMOVE":
        return state.filter(todo => todo.id !==
action.payload);
      default:
        return state;
    }
  };
  /* useReducer can take 3 args:
  1. Reducer function
  2. Starting state
  3. Can pass an initial action here
  */
  // We get back exactly 2 elements from useReducer
  // [] is the initial state passed to todoListReducer()
  const [todoList, dispatch] = useReducer(todoListReducer,
[]);
  /* After this line we can start dispatching actions.

```

```

This is the reason we have moved code at the top.

*/
// Pass a function to useEffect() as its 1st argument
useEffect(() => {
  axios.get("https://fir-
6cd72.firebaseio.com/todo.json").then(result => {
    // console.log(result);
    const todoData = result.data;
    const todos = [];
    for (const key in todoData) {
      todos.push({ id: key, name: todoData[key].name });
    }
    // Instead of the below statement, now we will dispatch
actions
    //setTodoList(todos);
    dispatch({ type: "SET", payload: todos });
  });
  return () => {
    console.log("cleanup");
  };
}, [todoName]);

const mouseMoveHandler = event => {
  console.log(event.clientX, event.clientY);
};

useEffect(() => {
  document.addEventListener("mousemove", mouseMoveHandler);
  // return is used to write clean up code
  return () => {
    // Remove the mousemove listener that executes
mouseMoveHandler
    document.removeEventListener("mousemove",
mouseMoveHandler);
  };
}, []);

```

```

useEffect(() => {
  if (submittedTodo) {
    // setTodoList(todoList.concat(submittedTodo));
    dispatch({ type: "ADD", payload: submittedTodo });
  }
}, [submittedTodo]);

const inputChangeHandler = event => {
  setTodoName(event.target.value);
};

const todoAddHandler = () => {
  axios
    .post("https://fir-6cd72.firebaseio.com/todo.json", {
      name: todoName
    })
    .then(res => {
      setTimeout(() => {
        const todoItem = { id: res.data.name, name: todoName };
        setSubmittedTodo(todoItem);
      }, 3000);
    })
    .catch(err => {
      console.log(err);
    });
};

return (
  <React.Fragment>
    <input
      type="text"
      placeholder="Todo"
      onChange={inputChangeHandler}
      value={todoName}
    />
    <button type="button" onClick={todoAddHandler}>
      Add
    </button>
  </React.Fragment>
);

```

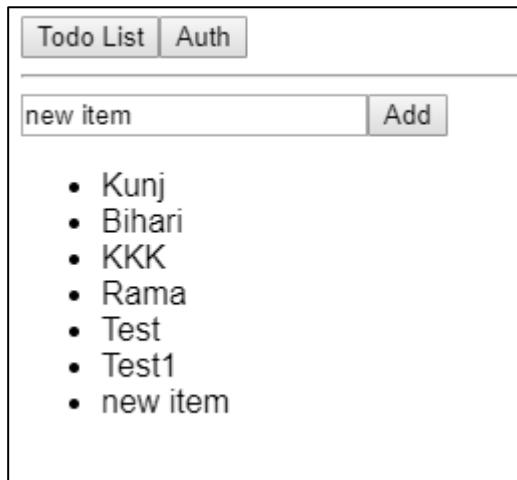
```

        </button>
      <ul>
        {todoList.map(todo => (
          <li key={todo.id}>{todo.name}</li>
        )));
      </ul>
    </React.Fragment>
  );
};

export default Todo;

```

O/P: ('new item' is added to the list after clicking 'Add')



17.useReducer() vs useState()

Delete items. Click on a list item and the item will be deleted in DB as well as front end.

Todo.js

```

import React, { useState, useEffect, useReducer } from
"react";
import axios from "axios";

const Todo = props => {
  const [todoName, setTodoName] = useState("");
  const [submittedTodo, setSubmittedTodo] = useState(null);

```

```

// As we are using useReducer. Comment this line of code
//const [todoList, setTodoList] = useState([]);

//React will pass these arguments automatically for us
// state -> latest state
// action -> Info about what to do
const todoListReducer = (state, action) => {
  switch (action.type) {
    case "ADD":
      return state.concat(action.payload);
      // this will set my state to a completely new list
    case "SET":
      return action.payload;
    case "REMOVE":
      return state.filter(todo => todo.id !==
action.payload);
    default:
      return state;
  }
};

/* useReducer can take 3 args:
1. Reducer function
2. Starting state
3. Can pass an initial action here
*/
// We get back exactly 2 elements from useReducer
// [] is the initial state passed to todoListReducer()
const [todoList, dispatch] = useReducer(todoListReducer,
[]);

/* After this line we can start dispatching actions.
This is the reason we have moved code at the top.
*/

// Pass a function to useEffect() as its 1st argument
useEffect(() => {
  axios.get("https://fir-
6cd72.firebaseio.com/todo.json") .then(result => {

```

```

// console.log(result);
const todoData = result.data;
const todos = [];
for (const key in todoData) {
  todos.push({ id: key, name: todoData[key].name });
}
// Instead of the below statement, now we will dispatch actions
//setTodoList(todos);
dispatch({ type: "SET", payload: todos });
});
return () => {
  console.log("cleanup");
};
}, [todoName]);

const mouseMoveHandler = event => {
  console.log(event.clientX, event.clientY);
};

useEffect(() => {
  document.addEventListener("mousemove", mouseMoveHandler);
  // return is used to write clean up code
  return () => {
    // Remove the mousemove listener that executes
    mouseMoveHandler
    document.removeEventListener("mousemove",
    mouseMoveHandler);
  };
}, []);

useEffect(() => {
  if (submittedTodo) {
    // setTodoList(todoList.concat(submittedTodo));
    dispatch({ type: "ADD", payload: submittedTodo });
  }
}, [submittedTodo]);

```

```

const inputChangeHandler = event => {
  setTodoName(event.target.value);
};

const todoAddHandler = () => {
  axios
    .post("https://fir-6cd72.firebaseio.com/todo.json", {
      name: todoName
    })
    .then(res => {
      setTimeout(() => {
        const todoItem = { id: res.data.name, name: todoName };
        setSubmittedTodo(todoItem);
      }, 3000);
    })
    .catch(err => {
      console.log(err);
    });
};

const todoRemoveHandler = todoId => {
  axios
    .delete(`https://fir-6cd72.firebaseio.com/${todoId}.json`)
    .then(res => {
      dispatch({ type: "REMOVE", payload: todoId });
    })
    .catch(err => console.log(err));
};

return (
  <React.Fragment>
    <input
      type="text"
      placeholder="Todo"
      onChange={inputChangeHandler}
      value={todoName}

```

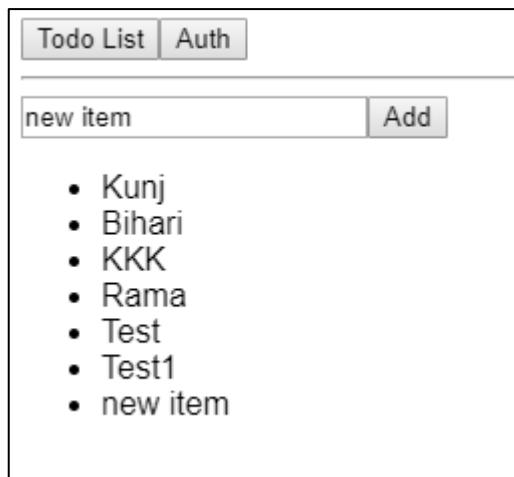
```

    />
    <button type="button" onClick={todoAddHandler}>
      Add
    </button>
    <ul>
      {todoList.map(todo => (
        <li key={todo.id}
        onClick={todoRemoveHandler.bind(this, todo.id)}>
          {todo.name}
        </li>
      )) }
    </ul>
  </React.Fragment>
);
};

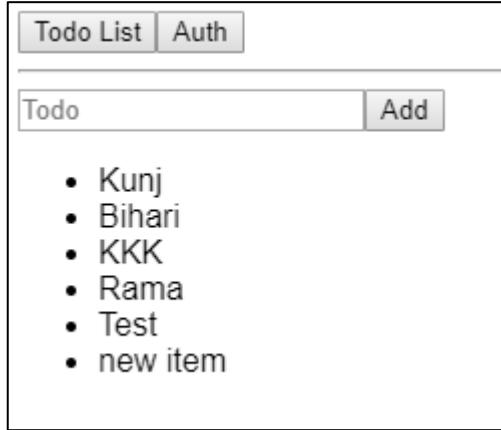
export default Todo;

```

O/P before deletion:



O/P after clicking 'Test1':



Now that we are using `useReducer` and `dispatch`, we can also simplify our `submittedTodo` logic. Comment the below line of codes in `Todo.js`,

```
const [submittedTodo, setSubmittedTodo] = useState(null);
```

And

```
useEffect(() => {
  if (submittedTodo) {
    // setTodoList(todoList.concat(submittedTodo));
    dispatch({ type: "ADD", payload: submittedTodo });
  }
}, [submittedTodo]);
```

In place of above code, we can just dispatch in `axios.post`.

`Todo.js`

```
import React, { useState, useEffect, useReducer } from "react";
import axios from "axios";

const Todo = props => {
  const [todoName, setTodoName] = useState("");
  //const [submittedTodo, setSubmittedTodo] = useState(null);
  // As we are using useReducer. Comment this line of code
  //const [todoList, setTodoList] = useState([]);
  //React will pass these arguments automatically for us
  // state -> latest state
  // action -> Info about what to do
  const todoListReducer = (state, action) => {
```

```

switch (action.type) {
  case "ADD":
    return state.concat(action.payload);
    // this will set my state to a completely new list
  case "SET":
    return action.payload;
  case "REMOVE":
    return state.filter(todo => todo.id !==
action.payload);
  default:
    return state;
}
};

/* useReducer can take 3 args:
1. Reducer function
2. Starting state
3. Can pass an initial action here
*/
// We get back exactly 2 elements from useReducer
// [] is the initial state passed to todoListReducer()
const [todoList, dispatch] = useReducer(todoListReducer, []);
/* After this line we can start dispatching actions.
This is the reason we have moved code at the top.
*/
// Pass a function to useEffect() as its 1st argument
useEffect(() => {
  axios.get("https://fir-
6cd72.firebaseio.com/todo.json").then(result => {
    // console.log(result);
    const todoData = result.data;
    const todos = [];
    for (const key in todoData) {
      todos.push({ id: key, name: todoData[key].name });
    }
    // Instead of the below statement, now we will dispatch
actions
    //setTodoList(todos);
    dispatch({ type: "SET", payload: todos });
  });
}

```

```

        return () => {
            console.log("cleanup");
        };
    }, [todoName]);

const mouseMoveHandler = event => {
    console.log(event.clientX, event.clientY);
};

useEffect(() => {
    document.addEventListener("mousemove", mouseMoveHandler);
    // return is used to write clean up code
    return () => {
        // Remove the mousemove listener that executes
        mouseMoveHandler
        document.removeEventListener("mousemove",
        mouseMoveHandler);
    };
}, []);

// useEffect(() => {
//     if (submittedTodo) {
//         // setTodoList(todoList.concat(submittedTodo));
//         dispatch({ type: "ADD", payload: submittedTodo });
//     }
// }, [submittedTodo]);

const inputChangeHandler = event => {
    setTodoName(event.target.value);
};

const todoAddHandler = () => {
    axios
        .post("https://fir-6cd72.firebaseio.com/todo.json", {
name: todoName })
        .then(res => {
            setTimeout(() => {
                const todoItem = { id: res.data.name, name:
                todoName };
                //setSubmittedTodo(todoItem);
                dispatch({ type: "ADD", payload: todoItem });
            });
        });
};

```

```

        }, 3000);
    })
    .catch(err => {
        console.log(err);
    });
};

const todoRemoveHandler = todoId => {
    axios
        .delete(`https://fir-
6cd72.firebaseio.com/${todoId}.json`)
        .then(res => {
            dispatch({ type: "REMOVE", payload: todoId });
        })
        .catch(err => console.log(err));
};

return (
    <React.Fragment>
        <input
            type="text"
            placeholder="Todo"
            onChange={inputChangeHandler}
            value={todoName}
        />
        <button type="button" onClick={todoAddHandler}>
            Add
        </button>
        <ul
            onClick={todoRemoveHandler.bind(this, todo.id)}>
            {todoList.map(todo => (
                <li key={todo.id}>
                    {todo.name}
                </li>
            )));
        </ul>
    </React.Fragment>
);
};

```

```
export default Todo;
```

18.Working with References and useRef()

While using class based components, we could use references to interact with any element on our page. In functional component this was not possible bcz they did not have the property that stored the references. Using hooks we can use references in functional component.

Todo.js

1) Import useRef hook

2) use 'ref' in the <input>

```
<input
  type="text"
  placeholder="Todo"
  // Use 'useRef' hook instead of the below 2
  attributes
  // onChange={inputChangeHandler}
  // value={todoName}
  ref={todoInputRef}
/>
```

3) get rid of 'useState'

```
//const [todoName, setTodoName] = useState("");
```

4)Comment 'inputChangeHandler' as we are not updating this value with 'onChange' anymore.

5)create todoName const and assign to it the latest value of the ref. ref refers to the <input> and <input> has a value field.

```
const todoName = todoInputRef.current.value;
```

Todo.js

```
import React, { useEffect, useReducer, useRef } from "react";
import axios from "axios";

const Todo = props => {
```

```

//const [todoName, setTodoName] = useState("");
//const [submittedTodo, setSubmittedTodo] = useState(null);
// As we are using useReducer. Comment this line of code
//const [todoList, setTodoList] = useState([]);
const todoInputRef = useRef();

//React will pass these arguments automatically for us
// state -> latest state
// action -> Info about what to do
const todoListReducer = (state, action) => {
  switch (action.type) {
    case "ADD":
      return state.concat(action.payload);
      // this will set my state to a completely new list
    case "SET":
      return action.payload;
    case "REMOVE":
      return state.filter(todo => todo.id !==
action.payload);
    default:
      return state;
  }
};

/* useReducer can take 3 args:
1. Reducer function
2. Starting state
3. Can pass an initial action here
*/
// We get back exactly 2 elements from useReducer
// [] is the initial state passed to todoListReducer()
const [todoList, dispatch] = useReducer(todoListReducer,
[]);

/* After this line we can start dispatching actions.
This is the reason we have moved code at the top.
*/

// Pass a function to useEffect() as its 1st argument

```

```

useEffect(() => {
  axios.get("https://fir-
6cd72.firebaseio.com/todo.json").then(result => {
    // console.log(result);
    const todoData = result.data;
    const todos = [];
    for (const key in todoData) {
      todos.push({ id: key, name: todoData[key].name });
    }
    // Instead of the below statement, now we will dispatch
actions
    //setTodoList(todos);
    dispatch({ type: "SET", payload: todos });
  });
  return () => {
    console.log("cleanup");
  };
}, []);

const mouseMoveHandler = event => {
  console.log(event.clientX, event.clientY);
};

useEffect(() => {
  document.addEventListener("mousemove", mouseMoveHandler);
  // return is used to write clean up code
  return () => {
    // Remove the mousemove listener that executes
mouseMoveHandler
    document.removeEventListener("mousemove",
mouseMoveHandler);
  };
}, []);

// useEffect(() => {
//   if (submittedTodo) {
//     // setTodoList(todoList.concat(submittedTodo));
//     dispatch({ type: "ADD", payload: submittedTodo });
//   }
// }, []);

```

```

//      }
// }, [submittedTodo]);

// const inputChangeHandler = event => {
//   setTodoName(event.target.value);
// };

const todoAddHandler = () => {
  const todoName = todoInputRef.current.value;
  axios
    .post("https://fir-6cd72.firebaseio.com/todo.json", {
      name: todoName
    })
    .then(res => {
      setTimeout(() => {
        const todoItem = { id: res.data.name, name: todoName };
        //setSubmittedTodo(todoItem);
        dispatch({ type: "ADD", payload: todoItem });
      }, 3000);
    })
    .catch(err => {
      console.log(err);
    });
};

const todoRemoveHandler = todoId => {
  axios
    .delete(`https://fir-
6cd72.firebaseio.com/${todoId}.json`)
    .then(res => {
      dispatch({ type: "REMOVE", payload: todoId });
    })
    .catch(err => console.log(err));
};

return (
  <React.Fragment>
    <input

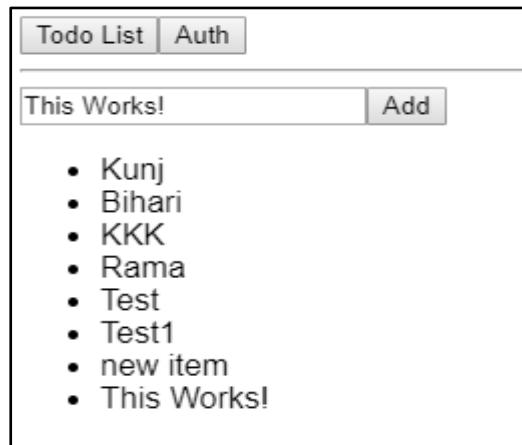
```

```

        type="text"
        placeholder="Todo"
        // Use 'useRef' hook instead of the below 2
        attributes
        // onChange={inputChangeHandler}
        // value={todoName}
        ref={todoInputRef}
      />
      <button type="button" onClick={todoAddHandler}>
        Add
      </button>
      <ul>
        {todoList.map(todo => (
          <li key={todo.id}
        onClick={todoRemoveHandler.bind(this, todo.id)}>
          {todo.name}
        </li>
        ) ) }
      </ul>
    </React.Fragment>
  ) ;
}
export default Todo;

```

O/P:



19. Preparing & Optimizing

Create List.js to outsource logic out of todo.js.

Todo.js

```
import React, { useEffect, useReducer, useRef } from "react";
import axios from "axios";
import List from "./List";

const Todo = props => {
    //const [todoName, setTodoName] = useState("");
    //const [submittedTodo, setSubmittedTodo] = useState(null);
    // As we are using useReducer. Comment this line of code
    //const [todoList, setTodoList] = useState([]);
    const todoInputRef = useRef();

    //React will pass these arguments automatically for us
    // state -> latest state
    // action -> Info about what to do
    const todoListReducer = (state, action) => {
        switch (action.type) {
            case "ADD":
                return state.concat(action.payload);
                // this will set my state to a completely new list
            case "SET":
                return action.payload;
            case "REMOVE":
                return state.filter(todo => todo.id !==
action.payload);
            default:
                return state;
        }
    };
    /* useReducer can take 3 args:
    1. Reducer function
    2. Starting state
    3. Can pass an initial action here
    */
}
```

```

// We get back exactly 2 elements from useReducer
// [] is the initial state passed to todoListReducer()
const [todoList, dispatch] = useReducer(todoListReducer,
[]);

/* After this line we can start dispatching actions.
This is the reason we have moved code at the top.
*/


// Pass a function to useEffect() as its 1st argument
useEffect(() => {
    axios.get("https://fir-
6cd72.firebaseio.com/todo.json").then(result => {
        // console.log(result);
        const todoData = result.data;
        const todos = [];
        for (const key in todoData) {
            todos.push({ id: key, name: todoData[key].name });
        }
        // Instead of the below statement, now we will dispatch
actions
        //setTodoList(todos);
        dispatch({ type: "SET", payload: todos });
    });
    return () => {
        console.log("cleanup");
    };
}, []);

const mouseMoveHandler = event => {
    console.log(event.clientX, event.clientY);
};

// useEffect(() => {
//     document.addEventListener("mousemove",
mouseMoveHandler);
//     // return is used to write clean up code
//     return () => {

```

```

//      // Remove the mousemove listener that executes
mouseMoveHandler
//      document.removeEventListener("mousemove",
mouseMoveHandler);
//  } ;
// }, []);

// useEffect(() => {
//   if (submittedTodo) {
//     // setTodoList(todoList.concat(submittedTodo));
//     dispatch({ type: "ADD", payload: submittedTodo });
//   }
// }, [submittedTodo]);

// const inputChangeHandler = event => {
//   setTodoName(event.target.value);
// };

const todoAddHandler = () => {
  const todoName = todoInputRef.current.value;
  axios
    .post("https://fir-6cd72.firebaseio.com/todo.json", {
name: todoName })
    .then(res => {
      setTimeout(() => {
        const todoItem = { id: res.data.name, name:
todoName };
        //setSubmittedTodo(todoItem);
        dispatch({ type: "ADD", payload: todoItem });
      }, 3000);
    })
    .catch(err => {
      console.log(err);
    });
};

const todoRemoveHandler = todoId => {

```

```

axios
    .delete(`https://fir-
6cd72.firebaseio.com/${todoId}.json`)
    .then(res => {
        dispatch({ type: "REMOVE", payload: todoId });
    })
    .catch(err => console.log(err));
};

return (
<React.Fragment>
<input
    type="text"
    placeholder="Todo"
    // Use 'useRef' hook instead of the below 2
attributes
    // onChange={inputChangeHandler}
    // value={todoName}
    ref={todoInputRef}
/>
<button type="button" onClick={todoAddHandler}>
    Add
</button>
{ /* <ul>
    {todoList.map(todo => (
        <li key={todo.id}
onClick={todoRemoveHandler.bind(this, todo.id)}>
            {todo.name}
        </li>
    )) }
</ul> */}
<List items={todoList} onClick={todoRemoveHandler} />
</React.Fragment>
);
};

export default Todo;

```

List.js

```
import React from "react";

const list = props => {
  console.log("Rendering the list...");

  return (
    <ul>
      {props.items.map(item => (
        <li key={item.id} onClick={props.onClick.bind(this, item.id)}>
          {item.name}
        </li>
      ))}
    </ul>
  );
};

export default list;
```

let us say we want to add validation to our input.

Todo.js

```
import React, { useState, useEffect, useReducer, useRef } from "react";
import axios from "axios";
import List from "./List";

const Todo = props => {
  const [inputIsValid, setInputIsValid] = useState(false);
  //const [todoName, setTodoName] = useState("");
  //const [submittedTodo, setSubmittedTodo] = useState(null);
  // As we are using useReducer. Comment this line of code
  //const [todoList, setTodoList] = useState([]);
  const todoInputRef = useRef();
```

```

//React will pass these arguments automatically for us
// state -> latest state
// action -> Info about what to do
const todoListReducer = (state, action) => {
  switch (action.type) {
    case "ADD":
      return state.concat(action.payload);
      // this will set my state to a completely new list
    case "SET":
      return action.payload;
    case "REMOVE":
      return state.filter(todo => todo.id !==
action.payload);
    default:
      return state;
  }
};

/* useReducer can take 3 args:
1. Reducer function
2. Starting state
3. Can pass an initial action here
*/
// We get back exactly 2 elements from useReducer
// [] is the initial state passed to todoListReducer()
const [todoList, dispatch] = useReducer(todoListReducer,
[]);

/* After this line we can start dispatching actions.
This is the reason we have moved code at the top.
*/


// Pass a function to useEffect() as its 1st argument
useEffect(() => {
  axios.get("https://fir-
6cd72.firebaseio.com/todo.json").then(result => {
    // console.log(result);
    const todoData = result.data;
    const todos = [];

```

```

        for (const key in todoData) {
            todos.push({ id: key, name: todoData[key].name });
        }
        // Instead of the below statement, now we will dispatch
        actions
        //setTodoList(todos);
        dispatch({ type: "SET", payload: todos });
    );
    return () => {
        console.log("cleanup");
    };
}, []);
}

const mouseMoveHandler = event => {
    console.log(event.clientX, event.clientY);
};

const inputValidationHandler = event => {
    if (event.target.value.trim() === "") {
        setInputIsValid(false);
    } else {
        setInputIsValid(true);
    }
};

// useEffect(() => {
//     document.addEventListener("mousemove",
mouseMoveHandler);
//     // return is used to write clean up code
//     return () => {
//         // Remove the mousemove listener that executes
mouseMoveHandler
//         document.removeEventListener("mousemove",
mouseMoveHandler);
//     };
// }, []);

```

```

// useEffect(() => {
//   if (submittedTodo) {
//     // setTodoList(todoList.concat(submittedTodo));
//     dispatch({ type: "ADD", payload: submittedTodo });
//   }
// }, [submittedTodo]);

// const inputChangeHandler = event => {
//   setTodoName(event.target.value);
// };

const todoAddHandler = () => {
  const todoName = todoInputRef.current.value;
  axios
    .post("https://fir-6cd72.firebaseio.com/todo.json", {
      name: todoName })
    .then(res => {
      setTimeout(() => {
        const todoItem = { id: res.data.name, name:
todoName };
        //setSubmittedTodo(todoItem);
        dispatch({ type: "ADD", payload: todoItem });
      }, 3000);
    })
    .catch(err => {
      console.log(err);
    });
};

const todoRemoveHandler = todoId => {
  axios
    .delete(`https://fir-
6cd72.firebaseio.com/${todoId}.json`)
    .then(res => {
      dispatch({ type: "REMOVE", payload: todoId });
    })
    .catch(err => console.log(err));
};

```

```

} ;

return (
<React.Fragment>
<input
  type="text"
  placeholder="Todo"
  // Use 'useRef' hook instead of the below 2
attributes
  // onChange={inputChangeHandler}
  // value={todoName}
  ref={todoInputRef}
  onChange={inputValidationHandler}
  style={{ backgroundColor: inputIsValid ?
"transparent" : "red" }}}
/>
<button type="button" onClick={todoAddHandler}>
  Add
</button>
{ /* <ul>
  todoList.map(todo => (
    <li key={todo.id}
onClick={todoRemoveHandler.bind(this, todo.id)}>
      {todo.name}
    </li>
  ) ) }
</ul> */}
<List items={todoList} onClick={todoRemoveHandler} />
</React.Fragment>
) ;
};

export default Todo;

```

Problem: The issue with the above code is that as we type into the todoList input, with each character typing the below statement in List.js gets executed.

```
console.log("Rendering the list...");
```

This is not something we want. Whenever you set some state in any component, react will re-render it. If that state does not directly affect the <List>, then we do not want <List> to re-render. The solution is in the next lecture.

20.Avoiding Unnecessary Re-Rendering

To stop <List> from unnecessary render, use '**useMemo**' hook.

Todo.js

```
import React, { useState, useEffect, useReducer, useRef,
useMemo } from "react";
import axios from "axios";
import List from "./List";

const Todo = props => {
  const [inputIsValid, setInputIsValid] = useState(false);
  //const [todoName, setTodoName] = useState("");
  //const [submittedTodo, setSubmittedTodo] = useState(null);
  // As we are using useReducer. Comment this line of code
  //const [todoList, setTodoList] = useState([]);
  const todoInputRef = useRef();

  //React will pass these arguments automatically for us
  // state -> latest state
  // action -> Info about what to do
  const todoListReducer = (state, action) => {
    switch (action.type) {
      case "ADD":
        return state.concat(action.payload);
        // this will set my state to a completely new list
      case "SET":
        return action.payload;
      case "REMOVE":
        return state.filter(todo => todo.id !==
action.payload);
      default:
```

```

        return state;
    }
};

/* useReducer can take 3 args:
1. Reducer function
2. Starting state
3. Can pass an initial action here
*/
// We get back exactly 2 elements from useReducer
// [] is the initial state passed to todoListReducer()
const [todoList, dispatch] = useReducer(todoListReducer,
[]);

/* After this line we can start dispatching actions.
This is the reason we have moved code at the top.
*/

// Pass a function to useEffect() as its 1st argument
useEffect(() => {
    axios.get("https://fir-
6cd72.firebaseio.com/todo.json").then(result => {
        // console.log(result);
        const todoData = result.data;
        const todos = [];
        for (const key in todoData) {
            todos.push({ id: key, name: todoData[key].name });
        }
        // Instead of the below statement, now we will dispatch
actions
        //setTodoList(todos);
        dispatch({ type: "SET", payload: todos });
    });
    return () => {
        console.log("cleanup");
    };
}, []);

const mouseMoveHandler = event => {

```

```

        console.log(event.clientX, event.clientY);
    };

const inputValidationHandler = event => {
    if (event.target.value.trim() === "") {
        setInputIsValid(false);
    } else {
        setInputIsValid(true);
    }
};

// useEffect(() => {
//   document.addEventListener("mousemove",
mouseMoveHandler);
//   // return is used to write clean up code
//   return () => {
//     // Remove themousemove listener that executes
//     mouseMoveHandler
//     document.removeEventListener("mousemove",
//     mouseMoveHandler);
//   };
// }, []);

// useEffect(() => {
//   if (submittedTodo) {
//     // setTodoList(todoList.concat(submittedTodo));
//     dispatch({ type: "ADD", payload: submittedTodo });
//   }
// }, [submittedTodo]);

// const inputChangeHandler = event => {
//   setTodoName(event.target.value);
// };

const todoAddHandler = () => {
    const todoName = todoInputRef.current.value;
    axios

```

```

    .post("https://fir-6cd72.firebaseio.com/todo.json", {
name: todoName })
    .then(res => {
      setTimeout(() => {
        const todoItem = { id: res.data.name, name:
todoName };
        //setSubmittedTodo(todoItem);
        dispatch({ type: "ADD", payload: todoItem });
      }, 3000);
    })
    .catch(err => {
      console.log(err);
    });
  };

const todoRemoveHandler = todoId => {
  axios
    .delete(`https://fir-
6cd72.firebaseio.com/${todoId}.json`)
    .then(res => {
      dispatch({ type: "REMOVE", payload: todoId });
    })
    .catch(err => console.log(err));
};

return (
  <React.Fragment>
    <input
      type="text"
      placeholder="Todo"
      // Use 'useRef' hook instead of the below 2
      attributes
      // onChange={inputChangeHandler}
      // value={todoName}
      ref={todoInputRef}
      onChange={inputValidationHandler}

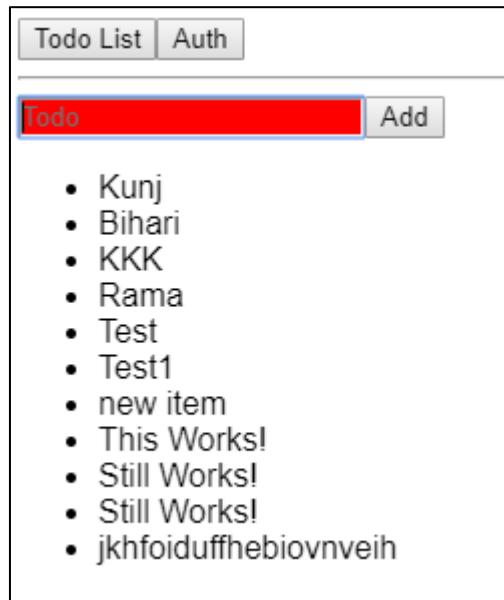
```

```

        style={ { backgroundColor: inputIsValid ? "white" : "#f0f0f0" } }
      >
      <button type="button" onClick={todoAddHandler}>
        Add
      </button>
      { /* <ul>
        {todoList.map(todo => (
          <li key={todo.id}
        onClick={todoRemoveHandler.bind(this, todo.id)}>
          {todo.name}
        </li>
        ) ) }
      </ul> */ }
      { /* If todoList does not change, react will have the
      cached value in the <List>. useMemo does this storage,
      retrieval, recreation, */}
      {useMemo(
        () => (
          <List items={todoList} onClick={todoRemoveHandler}>
        />
        ) ,
        [todoList]
      ) }
    </React.Fragment>
  ) ;
} ;
export default Todo;

```

O/P (Before start of typing input):



O/P (after starting typing in input field). We can observe no “Rendering the List...” message,

This screenshot illustrates the developer tools integrated with a browser window. On the left, the DevTools interface is visible, specifically the 'Console' tab which is selected. The console area is mostly empty, indicating no recent log output. On the right, the browser window displays the same 'Todo List' application. The input field now contains the partially typed text 'jfkelnhifeuhrcireu'. The list of todos remains the same as in the previous screenshot.

21. How to think about Functional Components with Hooks

22. Creating a Custom Hook

Create a new hook that deals with input validation.

Create a folder 'hooks' and a file 'forms.js' in it. **Functions that you are willing to use as hooks should start with 'use'.**

Forms.js

```

import { useState } from "react";

export const useFormInput = () => {
  const [value, setValue] = useState("");
  const [validity, setValidity] = useState(false);

  const inputChangeHandler = event => {
    setValue(event.target.value);
    if (event.target.value.trim() === "") {
      setValidity(false);
    } else {
      setValidity(true);
    }
  };

  return { value, onChange: inputChangeHandler,
  validity };
};

```

Todo.js

```

import React, { useState, useEffect, useReducer, useRef,
useMemo } from "react";
import axios from "axios";
import List from "./List";
import { useFormInput } from "../hooks/forms";

const Todo = props => {
  const [inputIsValid, setInputIsValid] = useState(false);
  //const [todoName, setTodoName] = useState("");
  //const [submittedTodo, setSubmittedTodo] = useState(null);
  // As we are using useReducer. Comment this line of code
  //const [todoList, setTodoList] = useState([]);
  //const todoInputRef = useRef();
  const todoInput = useFormInput();

```

```

//React will pass these arguments automatically for us
// state -> latest state
// action -> Info about what to do
const todoListReducer = (state, action) => {
  switch (action.type) {
    case "ADD":
      return state.concat(action.payload);
      // this will set my state to a completely new list
    case "SET":
      return action.payload;
    case "REMOVE":
      return state.filter(todo => todo.id !==
action.payload);
    default:
      return state;
  }
};

/* useReducer can take 3 args:
1. Reducer function
2. Starting state
3. Can pass an initial action here
*/
// We get back exactly 2 elements from useReducer
// [] is the initial state passed to todoListReducer()
const [todoList, dispatch] = useReducer(todoListReducer,
[]);

/* After this line we can start dispatching actions.
This is the reason we have moved code at the top.
*/


// Pass a function to useEffect() as its 1st argument
useEffect(() => {
  axios.get("https://fir-
6cd72.firebaseio.com/todo.json").then(result => {
    // console.log(result);
    const todoData = result.data;
    const todos = [];

```

```

        for (const key in todoData) {
            todos.push({ id: key, name: todoData[key].name });
        }
        // Instead of the below statement, now we will dispatch
        actions
        //setTodoList(todos);
        dispatch({ type: "SET", payload: todos });
    );
    return () => {
        console.log("cleanup");
    };
}, []);
}

const mouseMoveHandler = event => {
    console.log(event.clientX, event.clientY);
};

const inputValidationHandler = event => {
    if (event.target.value.trim() === "") {
        setInputIsValid(false);
    } else {
        setInputIsValid(true);
    }
};

// useEffect(() => {
//     document.addEventListener("mousemove",
mouseMoveHandler);
//     // return is used to write clean up code
//     return () => {
//         // Remove the mousemove listener that executes
mouseMoveHandler
//         document.removeEventListener("mousemove",
mouseMoveHandler);
//     };
// }, []);

```

```

// useEffect(() => {
//   if (submittedTodo) {
//     // setTodoList(todoList.concat(submittedTodo));
//     dispatch({ type: "ADD", payload: submittedTodo });
//   }
// }, [submittedTodo]);

// const inputChangeHandler = event => {
//   setTodoName(event.target.value);
// };

const todoAddHandler = () => {
  const todoName = todoInput.value;
  axios
    .post("https://fir-6cd72.firebaseio.com/todo.json", {
      name: todoName })
    .then(res => {
      setTimeout(() => {
        const todoItem = { id: res.data.name, name:
todoName };
        //setSubmittedTodo(todoItem);
        dispatch({ type: "ADD", payload: todoItem });
      }, 3000);
    })
    .catch(err => {
      console.log(err);
    });
};

const todoRemoveHandler = todoId => {
  axios
    .delete(`https://fir-
6cd72.firebaseio.com/${todoId}.json`)
    .then(res => {
      dispatch({ type: "REMOVE", payload: todoId });
    })
    .catch(err => console.log(err));
};

```

```

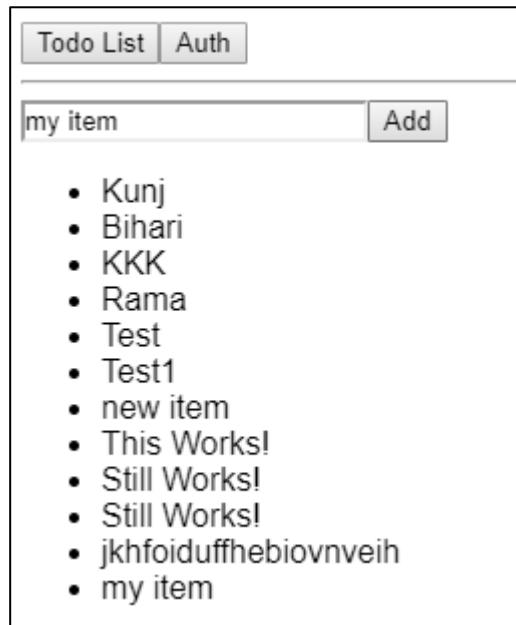
} ;

return (
  <React.Fragment>
    <input
      type="text"
      placeholder="Todo"
      onChange={todoInput.onChange}
      value={todoInput.value}
      style={ {
        backgroundColor: todoInput.validity === true ?
"transparent" : "red"
      } }
    />
    <button type="button" onClick={todoAddHandler}>
      Add
    </button>
    { /* <ul>
      {todoList.map(todo => (
        <li key={todo.id}
        onClick={todoRemoveHandler.bind(this, todo.id)}>
          {todo.name}
        </li>
      )) }
    </ul> */ }
    { /* If todoList does not change, react will have the
cached value in the <List>. useMemo does this storage,
retrieval, recreation, */ }
    {useMemo(
      () => (
        <List items={todoList} onClick={todoRemoveHandler}>
      ) ,
      [todoList]
    ) }
  </React.Fragment>
) ;

```

```
};  
export default Todo;
```

O/P:



23.Wrap Up