MERN and REACT.js

Introduction to Web Development

Web development stack is nothing but a set of tools typically used in tandem to develop web apps. It refers to the technologies that individual developer specializes in and use together to develop new pieces of software.

Web technology sets that include all the essential parts of a modern app are as follows: the **frontend framework**, the **backend solution** and the **database (relational or document-orinted)**

Introducton to stack:

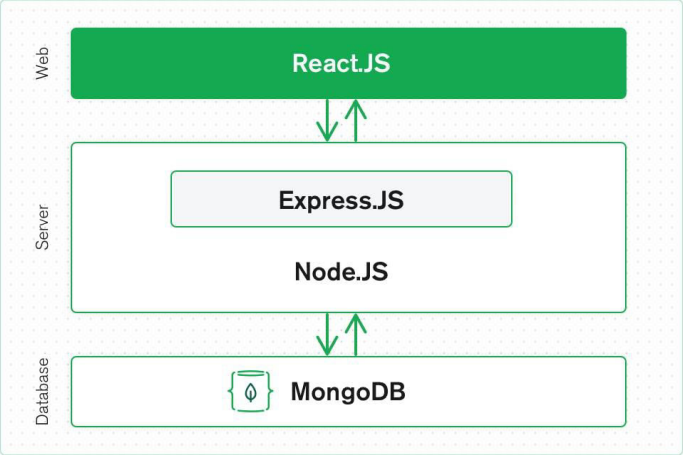
Any web application made by using multiple technologies. The combination of these technologies is called a **"stack,"** popularized by the LAMP stack, which is an acronym for Linux, Apache, MySQL, and PHP, which are all open-source components. As the web development world is continually changing, its technology stacks too changing. Top web development stacks are

- MEAN
- MERN
- Meteor.js
- Flutter
- The serverless Technology stack
- The LAMP technology stack
- Ruby on Rails Tech Stack

MERN stands for **MongoDB, Express, React, Node**

- MongoDB - Document database
- Express(.js) - Node.js web framework
- React(.js) - A client-side JavaScript library
- Node(.js) - The premier JavaScript web server

Allows you to easily construct a 3-tier architecture (frontend, backend, database) entirely using JavaScript and JSON.



Why MERN?

Ideally suited for web applications that have a large amount of interactivity built into the front-end.

- JavaScript Everywhere
- JSON Everywhere
- Isomorphic

REACT.JS

Free and open source front end JS Library for building UI and UI Components. Maintained by Facebook, a community of individual developers and companies. This can be used as base in the development of Single page applications and mobile applications and

is concerned with state management and rendering the state to DOM.

It is the declarative JavaScript Library for creating dynamic client-side applications

Builds up complex interfaces through simple Components, connect them to data on your backend server, and render them as HTML. Provides support for forms, error handling, events and render them as HTML.

Key points about React.js

- **Properties of React:**

  Declarative, Simple, Component based, Supports server side, Mobile support, Extensive, Fast , Easy to learn

- **Single way data flow**
  - A set of immutable values are passed to the components renderer as properties in its HTML tags. The component cannot directly modify any properties but can pass a call back function with the help of which we can do modifications.
  - This complete process is known as "**properties flow down; actions flow up**".

- **Virtual DOM**
  - Creates an in-memory data structure cache which computes the changes made and then updates the browser.
  - Allows a special feature that enables the programmer to code as if the whole page is rendered on each change whereas react library only renders components that actually change

Creation of React App

Can be done in two ways

- **Using node package manager(npm):** To setup a build environment for React that typically involved use of npm (node package manager), webpack, and Babel

- **Directly importing Reactjs library in HTML Code.** Defined in two .js files (**React** and **ReactDOM**)
  - <script src="https://unpkg.com/react@17/umd/react.development.js" crossorigin ></script>
  - <script src="https://unpkg.com/react-dom@17/umd/react-dom.development.js" crossorigin> </script>
  - The files differ for development and production.
    When deploying, replace "development.js" with "production.min.js"

React is used for handling the **view layer for web and mobile apps.** Allows developers to create **large web applications that can change data**, without reloading the page. Also allows to **create reusable UI components.** The main purpose of React is to **be fast, scalable, and simple. Works only on user interfaces** in the application

React Elements:

The browser DOM is made up of DOM elements. Similarly, the React DOM is made up of React elements. DOM elements and React elements may look the same, but they are actually quite different. A React element is a description of what the actual DOM element should look like. In other words, React elements are the instructions for how the browser DOM should be created. Syntax: React.createElement(type,{props},children);

The first one is the type of element we're creating, in this case an <h1> tag. This could also be another React component. Second is the properties list in the form of objects. Third argument is the content of the element used in the first argument.

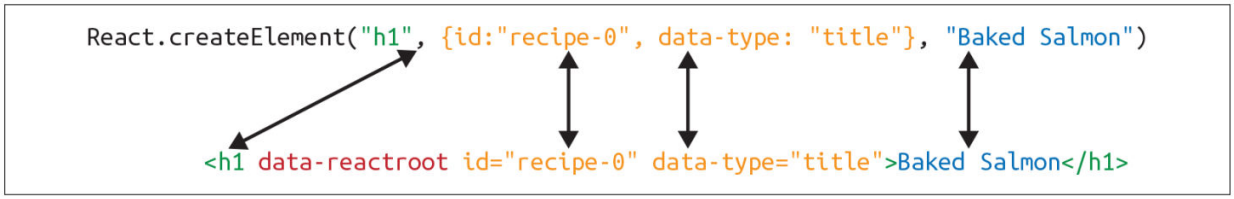
We can create a React element to represent an h1 using React.createElement

```
React.createElement("h1", null, "Baked Salmon")
```

When an element has attributes, they can be described with properties. Here is a sample of an HTML h1 tag that has id and data-type attributes.

```
React.createElement("h1", {id: "recipe-0", 'data-type': "title"}, "Baked Salmon")
```

Relationship between createElement and the DOM element



Note: data-reactroot will always appear as an attribute of the root ele- ment of your React component

React Component in brief:[In detail in L2 and L3]

A user control that has **code to represent visual interfaces and data.** An isolated piece of code which can be reused in one or the other module **.**Contains a root component in which other subcomponents are included. 2 types of components in React.js can be created.

- **Stateless Functional Component**
  - Includes simple JavaScript functions and immutable properties, i.e., the value for properties cannot be changed.

```
function Demo(props) {
  return <h1> Welcome to REACT JS, {props.Name} </h1>;
}
```

- **Stateful Class Component**
  - Classes which extend the Component class from React library. The class component must include the render method which returns HTML.

```
class Demo extends React.Component{
        render(){    return <h1>
                    Welcome to REACT JS, {props.Name} </h1>;    }
}
```

Calling/rendering the components:

ReactDOM: Contains the tools necessary to render React elements in the browser. All the tools necessary to generate HTML from the virtual DOM are found in this library.

ReactDOM.render() is responsible for rendering a React component. The first

parameter is the component class name. Second parameter is the destination where the component is to be rendered.

```
ReactDOM.render(
    React.createElement(Demo, null, null),
    document.getElementById('root')
);
```

Coding Example 1: Serverless Hello world: Using react, displaying a simple page on the browser

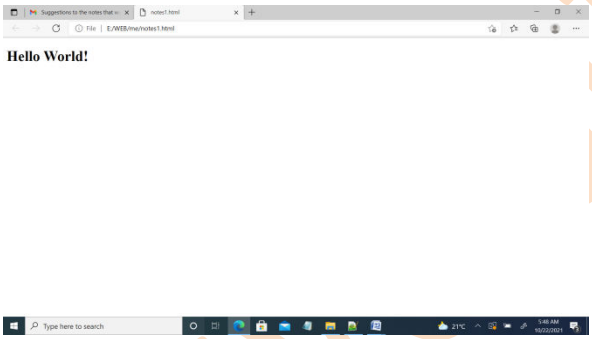
```
<!DOCTYPE HTML>
<html>
<head>
    <script crossdomain
src="https://unpkg.com/react@16/umd/react.development.js"> </script>
    <script crossdomain src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"> </script>
    <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js">
    </script>
</head>
<body>
  <div id="contents"></div><!-- this is where our component will appear -->
  <script type="text/babel">    // Important
      var contentNode = document.getElementById('contents');
      var component = <h1>Hello World!</h1>; // A simple JSX component
      ReactDOM.render(component, contentNode);
                                  // Render the component inside the content Node
  </script>
</body>
</html>
```

In the above code, <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js">

</script> , this is the **babel library which is a JSX transformer**. Then, there is the type of the script that you specify in the wrapping <script> tags around the JSX ode. The browser-based JSX compiler looks for all inline scripts of type “text/babel” and compiles the contents into the corresponding JavaScript. The other two scripts, react.js and react-dom.js, are the core React libraries that handle react component creation and rendering

```
<script type="text/babel">
```

Screenshot of example code 1 output:

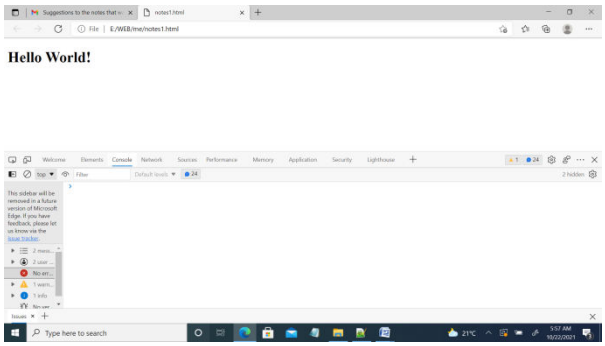


Coding Example 2: Multiple component calls/renders will render only the last one

In the above code, add these below lines and oberve the output

```
ReactDOM.render(component, contentNode);
ReactDOM.render(component, contentNode);
ReactDOM.render(component, contentNode);
ReactDOM.render(component, contentNode);
```

Screenshot of example code 1 output:



Including the JSX Code:

JSX uses a special syntax which allows you to mix HTML with JavaScript. JSX is a JavaScript XML used in React applications. An extension to JavaScript. Uses HTML syntax to create elements and components. Has tag name, attributes, and children. JSX compiles the code into pure JavaScript which can be understood by the browser.

Include the library:

```
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"> </script>
<script type="text/babel">
                ReactDOM.render(<h1>Welcome     to REACTJS</h1>,
                document.getElementById('root')
  );
```

Babel:

A JavaScript compiler that can translate markup or programming languages into JavaScript. Available for different conversions. React uses Babel to convert JSX into JavaScript.

Using React, print welcome to REACTJS on the web page. Use JSX and NonJSX both

Case 1: Using JSX Syntax

```
<html>
<head>
<script crossdomain    src="https://unpkg.com/react@16/umd/react.development.js">
</script>
<script crossdomain src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"> </script>
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"> </script>
</head>
<body>
        <div id = "root"></div>
        <script type = "text/babel">
```

```
        ReactDOM.render(<h1>welcome</h1>,document.getElementById("root"))
        </script>
</body>
</html>
```

Case 2: Using Non-JSX Syntax: small change in the above code

```
<html>
<head>
<script crossdomain    src="https://unpkg.com/react@16/umd/react.development.js">
</script>
<script crossdomain src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"> </script>
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"> </script>
</head>
<body>
        <div id = "root"></div>
        <script type = "text/babel">
        ReactDOM.render(React.createElement("h1","null","welcome to
reactJS"),document.getElementById("root"))
        </script>
</body>
</html>
```

References:

i) What is the use of React.createElement ? - GeeksforGeeks

ii) Textbooks from the syllabus copy

Components and Propertties

Components

It defines the **visuals and interactions** that make up what people see in app. Declares **how the view looks like, given the data**. When the data changes, if you are used to the jQuery way of doing things, you'd typically do some DOM manipulation. But the React doesn't do anything like that! The React library figures out how the new view looks, and just applies the changes between the old view and the new view. This makes the views consistent, predictable, easier to maintain, and simpler to understand.

Why Components?

There may be elements which are similar . There might be a need to make a change that will result in changes in multiple places. Similar to functions, if we could write code related to the element in one place, changes can be minimized. Solution is to have reusable piece of JavaScript code that output (via JSX) HTML elements

Types of Components

1. **Stateless Functional Component –** [Will be discussed later]
    - Includes simple JavaScript functions and immutable properties, i.e., the value for properties cannot be changed.
    - Use hooks to achieve functionality for making changes in properties using JS.
    - Used mainly for UI.

2**. Stateful Class Component**
    - Classes which extend the Component class from React library. The class component must include the render method which returns HTML.

First Stateful component

Creation of a component:

```
class HelloWorld extends React.Component
{       render()
        {   return <p>Hello, componentized world!</p>; }
}
```

The render() method is something that that the React calls when it needs to display the component. There are other methods with special meaning to React that can be implemented, called the Lifecycle functions, which provide hooks into various stages of the component formation and events. This will be discussed later. But render() is one that must be present; otherwise you'll have a component that has no screen presence.

Calling/Rendering a component:

Case 1: Add the JSX in the render method with a element with the tag name as the Component name

```
ReactDOM.render( <HelloWorld/>,
        document.querySelector("#container")
);
```

Case 2: Add the NON-JSX in the render method with a element using React.createElement

```
ReactDOM.render( React.createElement("HelloWorld",null, null),
        document.querySelector("#container")
);
```

Parameterized components:

Passing the attributes to the components during rendering and using these attributes as properties inside the component.

Properties

Properties are ways in which React components can be customized. Immutable and same as what attributes in HTML elements. Props are arguments passed into React components and are passed via HTML attributes. 2 steps to add properties to components

- Make the function of your component read the props from the props parameter
  Place the props inside curly brackets – { }. In JSX, if you want something to get evaluated as an expression, you need to wrap that something inside curly brackets. If you don't do that, the raw text gets printed out.

  return <h1>Hello, **{this.props.greetingtarget}**</h1>

- **Modify the Component Call** : When rendering the component, add the prop to the component using the attribute

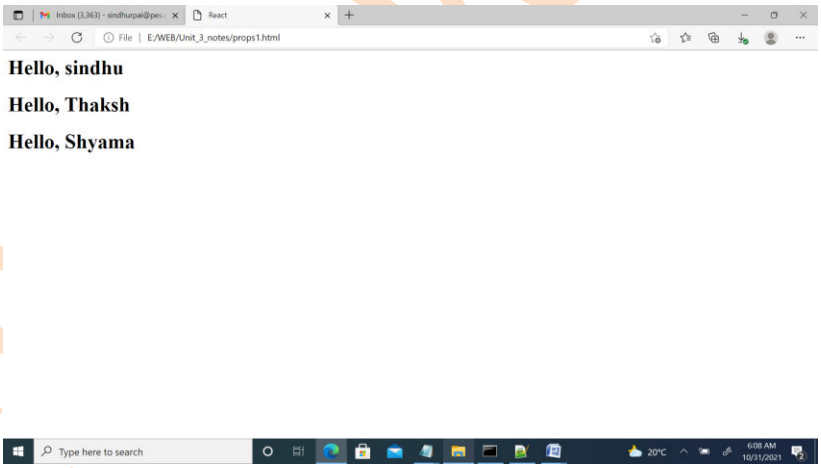
  <Helloworld greetingtarget = "sindhu" />

Coding example 1: Usage of this.props

```
<html>
    <head>
        <title>React</title>
        <script crossdomain
src="https://unpkg.com/react@16/umd/react.development.js"> </script>
        <script crossdomain src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"> </script>
        <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js">
</script>
    </head>
    <body>
        <div id="container"></div>
        <script type="text/babel">
            var dest = document.querySelector("#container");
            class Helloworld extends React.Component {
                render(){
                    return <h1>Hello, {this.props.greetingtarget}</h1>
                    }
```

```
                }
                ReactDOM.render(
                        <div>
                                <Helloworld greetingtarget = "sindhu" />
                                <Helloworld greetingtarget = "Thaksh" />
                                <Helloworld greetingtarget = "Shyama" />
                        </div>,
                        document.getElementById("container")
                );
        </script>
    </body>
</html>
```

Output:



Usage of this.props.children

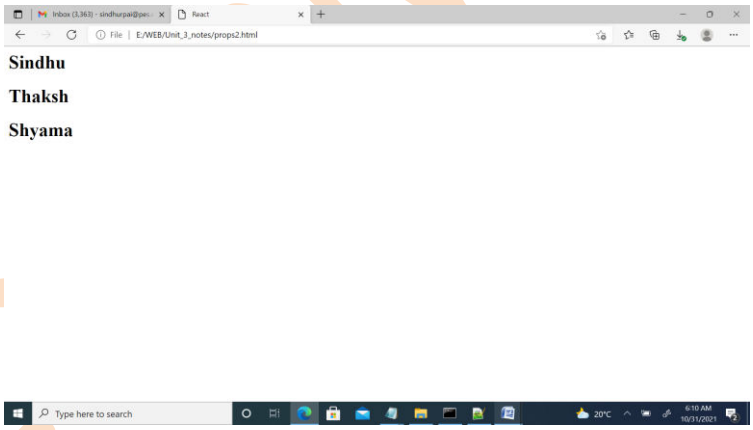
A special property that is passed to components automatically. It is used to display the data between the opening and closing JSX tags when invoking a component. Can have one element, multiple elements, or none at all. It's value will be respectively a single child node, an array of child nodes or undefined

Coding example 2:

```
<script type="text/babel">
```

```
class Helloworld extends React.Component {
        render(){
                return  <h1>{this.props.children}</h1>
                }
    }
ReactDOM.render(
        <div>
                <Helloworld>Sindhu</Helloworld>
                <Helloworld>Thaksh</Helloworld>
                <Helloworld>Shyama</Helloworld>
        </div>,
        document.querySelector("#container")
);
</script>
```

Output:



Validating the property values

The properties being passed to component to can be validated against a specification. This specification is supplied in the form of a static object called propTypes in the class, with the name of the property as the key and the validator as the value, which is one of the many constants exported by React.PropTypes, for example,

React.PropTypes.string

```
Helloworld.propTypes = { greetingtarget: React.PropTypes.string.isRequired,
                         greetingtarget_id: React.PropTypes.number
                       };
```

Property validation is checked only in development mode, and a warning is shown in the console when any validation fails. Since we are in an early stage in the development of the application, we expect more changes to the properties. You can also default the property values when the parent does not supply the value. For example, if you want the greeting_target to be defaulted to something else, rather than show an empty string, you can do this: `Helloworld.defaultProps = { greetingtarget: '-- no title --', };`

Few Points to think:

- Can we have nested components?
- Can we provide style to components?
- How to send properties from one component to another?
- What are the different methods under life cycle of components?

  Refer to the further notes for more details on the above points

Styling the components and Complex Components

Styling the components

There are different ways to style the components in React.

- Inline CSS – using the style attribute
- CSS in JS - Using Libraries JSS and Styled Components
- CSS Modules - css loader and Sass & SCSS
- Stylable

Inline CSS in Detail

Consider the below code to provide red color to the contents of h1. But this is not a react way of providing the styling to the components.

```
render()
{
    return (<h1 style = {color:"red"}>welcome to styling to components</h1>)
}
```

We create objects of style and render it inside the components in style attribute using the React technique. Let us understand this through an example code.

```
<body>
    <div id = "root"></div>
    <script type = "text/babel">
        class Letter extends React.Component{
            render() {
 //Object letterstyle contains all key-value pairs of styling  to be applied to letter
                var letterstyle = {
                marginRight: "10px", // observe this comma
                textAlign: "center",backgroundColor:"blue", //observe the camelcase
                color:"olive",  padding: "20px", display:"inline"      }
                return <h1 style = {letterstyle}> {this.props.children}</h1>
            //Setting the style attribute to refer to the object
```

```
                    }
}
            ReactDOM.render(<div><Letter>A</Letter>
                    <Letter>E</Letter>
                    <Letter>I</Letter>
                    <Letter>O</Letter>
                    <Letter>U</Letter>    </div>
                    ,document.getElementById("root"))
        </script>
    <body>
```

If we execute the above code, All letters will have the same background color. To avoid this, background color must be sent suring the component rendering.

```
<script type = "text/babel">
        class Letter extends React.Component{
                render() {
 //Object letterstyle contains all key-value pairs of styling  to be applied to letter
                    var letterstyle = {
                    marginRight: "10px", // observe this comma
                    textAlign: "center", backgroundColor: this.props.bgcolor,
                    color:"olive",   padding: "20px"         }
                    return <h1 style = {letterstyle}> {this.props.children}</h1>
                    }
        }
        ReactDOM.render(<div><Letter bgcolor = "pink"> A</Letter>
                    <Letter bgcolor = "red">E</Letter>
                    <Letter bgcolor = "#0000FF">I</Letter>
                    <Letter bgcolor = "#BBEE00">O</Letter>
                    <Letter bgcolor = "#EE00BB">U</Letter>          </div>
                    ,document.getElementById("root"))
        </script>
```

Output of above code is as below:

Advantage of creation of style object is , it provides flexibility to add more styling properties as and when required.

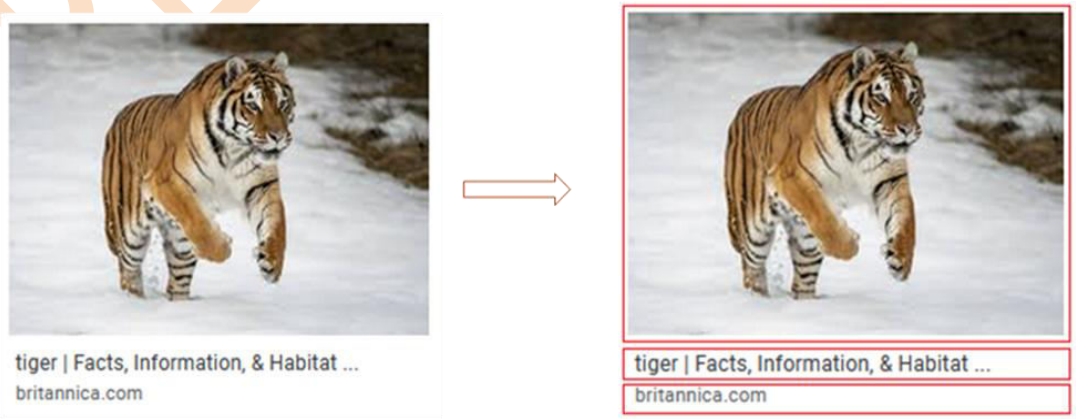
Complex Components

It is nothing but **building the component that uses other user defined components.** Also known as **component composition. React** lets you split the UI into smaller independent pieces so that you can reason about each piece in isolation. Using components rather than building the UI in a monolithic fashion also encourages reuse.. A component takes inputs (called properties) and its output is the rendered UI of the component. We will put together fine-grained components to build a larger UI.

Approach followed to build complex components:

- Identify the major visual elements
- Breaking them into individual components

Consider this example:



The Main complex component consists of **Image rendering, Caption rendering and Link rendering** on the web page.

The code for Main complex component is as below.

```
class Main extends React.Component{
        render() {
                return(<div>
                        <ReactImage />
                        <ReactCaption/>
                        <ReactLink/>
                </div>)
        }
}
```

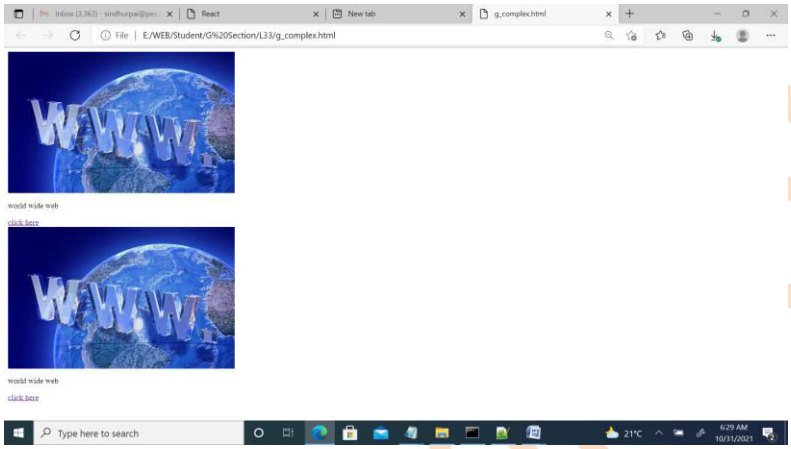
Codes for ReactImage, ReactCaption and ReactLink is as below.

```
class ReactImage extends React.Component{
        render(){
                return (<img src = "www.jfif"/>)
        }

class ReactCaption extends React.Component{
        render(){
                return (<p>world wide web</p>)
        }
class ReactLink extends React.Component{
        render(){
                return (<a href = "https://www.google.com" >click here</a>)
        }
```

When we render this using ReactDOM.render,

ReactDOM.render(** <Main/><Main/> ,** document.getElementById("root"))

, we get same image, caption and link both the times. This is because no attributes are sent during the component call. Also, clicking on the link, navigate to the same page as mentioned in the code.



Refer to the below code to have dynamic values for components. This adds this.props to access the values sent during the component call

```
class ReactImage extends React.Component{
        render()
        {       return (<img src = {this.props.src}/>)          }
}
class ReactCaption extends React.Component{
        render()
        {       return (<p>{this.props.caption}</p>)                    }
}
class ReactLink extends React.Component{
        render()
        {return (<a href = {this.props.href} >{this.props.link}</a>)          }
}
ReactDOM.render(<div>
        <Main  src  =  "www.jfif"  caption  =  "world"  href  =
```

"https://www.yahoo.com" link = "click here for yahoo"/>

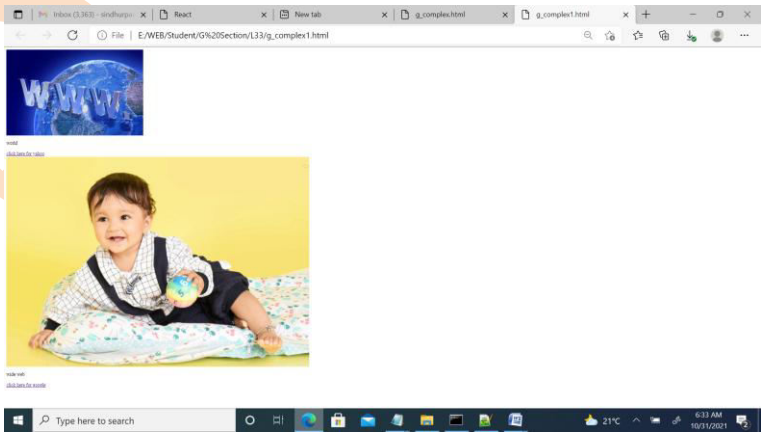
<Main src = "thaksh.jpg" caption = "wide web" href = "https://www.google.com" link = "click here for google"/> , document.getElementById("root"))

Transferring the properties:

If we keep the ReactDOM.render code same as previous, code doesn't throw any error. But no output as well. So think about where we have gone wrong. Observe that we have to **transfer the properties from Main component to three of the other components**. This is possible using the **spread operator(…)**

```
class Main extends React.Component{
        render(){   return(<div>
                    <ReactImage {...this.props}/>
                    <ReactCaption {...this.props}/>
                    <ReactLink {...this.props}/>
                </div>)  }
}
```

Output: Observe that rendering the same component with different values is possible using the attributes and props.



Think about having the same size for rendered images..!!

Component States and Life Cycle Methods

Introduction to Component States

The state is an instance of React Component Class can be defined as an object of a set of observable properties that control the behavior of the component. In other words, the State of a component is an object that holds some information that may change over the lifetime of the component. t **stores a component's dynamic data** and determines the component's behaviour. Because state is dynamic, it enables a component to keep track of changing information in between renders and for it to be dynamic and interactive.

Defining the state

State can only be accessed and modified inside the component and directly by the component. Must first have some **initial state**, should define the state in the constructor of the component's class.

```
class MyClass extends React.Component {
        constructor(props)  {
        super(props);  this.state = { attribute : "value" };    }
}
```

The state object can contain as many properties as you like

```
class MyClass extends React.Component {
        constructor(props)  {
        super(props);   this.state = { attribute1 : "value1", attribute2 : "value2",
attribute3 : "value3"};    }              }
```

Changing the state

State should **never be updated explicitly**. Must use **setstate().** It takes a single parameter and expects an object which should contain the set of values to be updated. Once the update is done the method implicitly calls the render() method to repaint the page.

```
this.setState({attributen : "valuen", attribute2 : "newvalue1", })
```

Differences between state and props

- States are mutable and Props are immutable
- States can be used in Class Components, Functional components with the use of React Hooks (useState and other methods) while Props don't have this limitation.
- Props are set by the parent component. State is generally updated by event handlers

Example code 1: Changing the state based on the click on the button

```
<body>
<div id = "root"> </div>
<script type = "text/babel">
    class Hello extends React.Component {
        constructor(props)      {
                super(props)
                this.state = {name:"sindhu",address:"nagarbavi",phno:"9876554"     }
                //this.updatestate = this.updatestate.bind(this) // if the function has no =>
        }
        render(){
                return (<div><h1>hello {this.state.name}</h1>
                <p> u r from {this.state.address} and ua contact number is
                {this.state.phno}</p>
                <button onClick = {this.updatestate}>click here to change state</button>
                </div> )
        }
        updatestate =()=>       {
                this.setState({name:"pai"})         }
    }
  ReactDOM.render(<Hello/>, document.getElementById("root"))
</script> </body>
```

Example code 2: Generate the Digital Clock using ReactJS

```
<script type = "text/babel">
        class Clock extends React.Component{
        constructor(){
                super()
                this.state = {time: new Date()}
                //this.tick = this.tick.bind(this)
        }
        //tick()
        tick = () => {          this.setState({time:new Date()})          }
        render(){   setInterval(this.tick,1000)
                    return (<h1>{this.state.time.toLocaleTimeString()}</h1>)
        }
       }
        ReactDOM.render(<Clock/>, document.getElementById("root"))
</script>
```

toLocaleTimeString(): Returns the time portion of a Date object as a string, using locale conventions.

Few points to think:

- What happens if you use setInterval directly inside class outside all functions?
- Calling the setInterval in render() is not a good idea as conventionally render function is used only to render the component . Then which is the best place to have this setInterval function call?

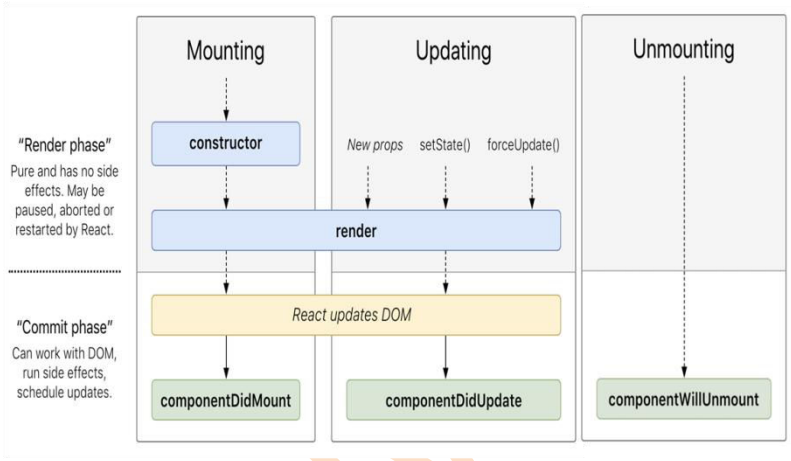
Life Cycle Methods

The series of events that happen from the starting of a React component to its ending. Every component in React should go through the following lifecycle of events.

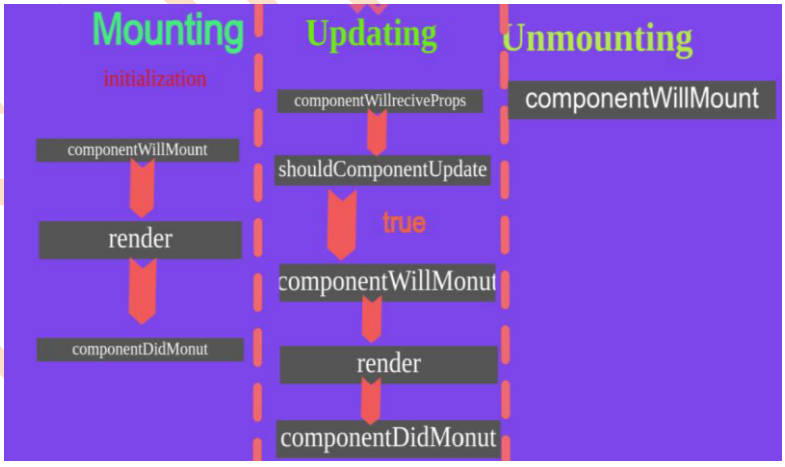
Mounting - Birth of the Component

Updating- Growing of component

Unmounting- End of the component



Functions defined in every phase is more clear with the below diagram



Functions/Methods in detail

1. constructor() : Premounting

This function is called before the component is mounted. Implementation requires calling of super() so that we can execute the constructor function that is inherited from

React.Component while adding our own functionality. Supports Initializing the state and binding our component

```
        constructor() {
  super()
  this.state = {
    key: "value"
  }
}
```

It is possible to use the constructor to set an initial state that is dependent upon props. Otherwise, this.props it will be undefined in the constructor, which can lead to a major error in the application.

```
constructor(props) {
  super(props);
  this.state = {
    color: props.initialColor
  };
}
```

2. componentWillMount()

This is called only once in the component lifecycle, immediately before the component is rendered. Executed before rendering, on both the server and the client side. Suppose you want to keep the time and date of when the component was created in your component state, you could set this up in componentWillMount.

```
componentWillMount() {
  this.setState({ startDateTime: new Date(Date.now())});
}
```

3. render()

Most useful life cycle method as it is the only method that is required. Handles the rendering of component while accessing this.state and this.props

4. **componentDidMount()**

Function is called once only, but immediately after the render() method has taken place. That means that the HTML for the React component has been rendered into the DOM and can be accessed if necessary. This method is used to perform any DOM manipulation of data-fetching that the component might need.

The best place to initiate API calls in order to fetch data from remote servers. Use setState which will cause another rendering but It will happen before the browser updates the UI. This is to ensure that the user won't see the intermediate state. AJAX requests and DOM or state updates should occur here. Also used for integration with other JavaScript frameworks like Node.js and any functions with late execution such as **setTimeout** or **setInterva**l

5. componentWillReceiveProps()

Allows us to match the incoming props against our current props and make logical. We get our current props by calling this.props and the new value is the nextProps argument passed to the method. It is invoked as soon as the props are updated before another render method is called.

6. **shouldComponentUpdate()**

Allows a component to exit the Update life cycle if there's no reason to use a replacement render. It may be a no-op that returns true. Means while updating the component, we'll re-render.

7. componentWillUpdate()

Called just before the rendering

8. componentDidUpdate()

Is invoked immediately after updating occurs. Not called for the initial render. Will not be invoked if **shouldcomponentUpdate() returns false**.

9. **componentWillUnmount()**

The last function to be called immediately before the component is removed from the DOM. It is generally used to perform clean-up for any DOM-elements or timers created in componentWillMount.

```
componentWillUnmount() {
  clearInterval(this.interval);
}
```

Example code 3: Sequence of execution of life cycle methods

```
<script type = "text/babel">
        class Hello extends React.Component{
                constructor(props){
                        console.log("in constructor")
                        super(props)
                        this.state={ame:"sindhu",address:"nagarbavi",phno:"9876"}
                        //this.updatestate = this.updatestate.bind(this)
                        //this.fun1 = this.fun1.bind(this)
                }
                render(){
                        console.log("in render")
                        return (<div><h1>hello {this.state.name}</h1>
                                <p> u r from {this.state.address} and ua contact number is
{this.state.phno}</p>
                        <button onClick = {this.updatestate}>click here to change
state</button>
                        <button onClick = {this.fun1}> click here to delete the
user</button></div>
                        )
                }
                componentDidUpdate(prevProps, prevState){
                                console.log("component did update")
                }
                UNSAFE_componentWillUpdate()  {
                                console.log("will update")
                }
                componentDidMount(){
                                console.log("did mount")
                }
                UNSAFE_componentWillMount()  {
```

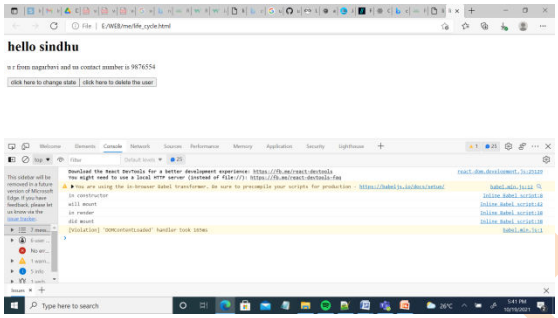
```
                    console.log("will mount")
            }
            fun1 = () =>    {
ReactDOM.unmountComponentAtNode(document.getElementById("root"))
            }
            componentWillUnmount(){
                    console.log("will unmount")
            }
            shouldComponentUpdate(nextProps, nextState) {
                console.log("yes, in shouldCOmponentupdate")
                return true;
            }
            updatestate =()=>{
                console.log("in updatestate")
                this.setState({name:"pai"})    }
            }
            ReactDOM.render(<Hello/>, document.getElementById("root"))
</script>
```

ReactDOM.unmountComponentAtNode(node to be deleted) : Used to delete the componenet from the DOM

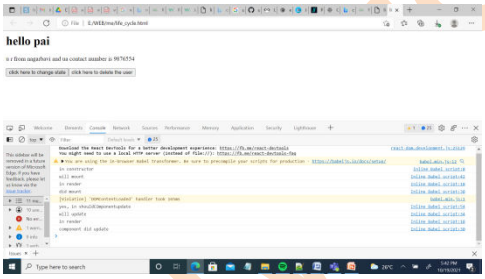
Note: In React v16.3, componentWillMount(), componentWillReceiveProps(), and componentWillUpdate() are marked as unsafe legacy lifecycle methods for deprecation process. They have often been misused and may cause more problems with the upcoming async rendering. As safer alternatives for those methods, **getSnapshotBeforeUpdate() and getDerivedStateFromProps()** were newly added. Since the roles of the unsafe methods and the newly added methods may overlap, React prevents the unsafe methods from being called when the alternatives are defined and outputs a **warning** message in the browser. So, explicitly UNSAFE_ keyword is used in redefining these methods.

Observe the output of the above code in console at every stage. Snapshot is attached in sequence.

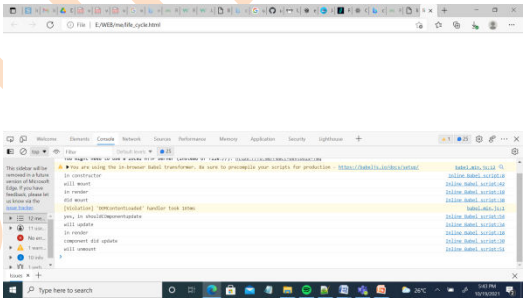
stage 1: once the page is loaded on the browser



Stage 2: when the first button is clicked



Stage 3: When the second button is clicked.



Example code 4: Digital clock using life cycle method

```
<body>
  <div id = "root"> </div>
  <script type = "text/babel">
        class Clock extends React.Component{
              constructor()   {
```

```
            super()
            this.state = {time: new Date()}
            this.tick = this.tick.bind(this)
            console.log("in constructor")
        }
        tick(){
            console.log("in tick")
            this.setState({time:new Date()})
        }
        render(){
            console.log("in render")
            return (<h1>{this.state.time.toLocaleTimeString()}</h1>)
        }
        componentDidMount(){
            console.log("in did mount")
            setInterval(this.tick,1000) // Observe that added here in didmount
method of lifecycle
        }
    }
    ReactDOM.render(<Clock/>, document.getElementById("root"))
  </script>
</body>
```

References

- ReactJS | State in React - GeeksforGeeks
- React Component Mounting And Unmounting - Learn.co
- Component Lifecycle | Build with React JS
- Rule | DeepScan

Stateless Components

Introduction

There exist components which do not use state. Such components just print out what is given to them via props, or they just render the same thing. For performance reasons and for clarity of code, it is recommended that such components (those that have only render()) are written as functions rather than classes: a function that takes in props and just renders based on it. It's as if the component's view is a pure function of its props, and it is stateless. The render() function itself can be the component. If a component does not depend on props, it can be written as a simple function whose name is the component name.

Stateless components are those components which don't have any state at all, which means you can't use this.setState inside these components.It has no lifecycle, so it is not possible to use lifecycle methods. When react renders our stateless component, all that it needs to do is just call the stateless component and pass down the props.

Note: A functional component is always a stateless component, but the class component can be stateless or stateful.

When would you use a stateless component??

- When you just need to present the props
- When you don't need a state, or any internal variables
- When creating element that does not need to be interactive
- When you want reusable code

When would you use a stateful component?

- When building element that accepts user input
- Element that is interactive on page
- When dependent on state for rendering, such as, fetching data before rendering
- When dependent on any data that cannot be passed down as props

Ways of creation of stateless components:

- The first is the ES2015 arrow function style with only the return value as an expression. There are no curly braces, and no statements, just a JSX expression

```
…
    const IssueRow = (props) => ( …)
…
```

- second style, a little less concise, is needed when the function is not a single expression. The main difference is the use of curly braces to indicate that there's going to be a return value, rather than the expression within the round braces being an implicit return of that expression's result.

```
…
    function IssueTable(props) {
        …
    }
…
```

Coding Example 1:Simple code illustration of stateless components

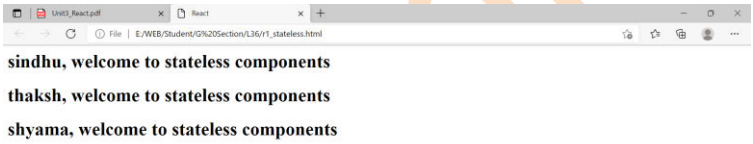
```
<div id = "root"></div>
<script type = "text/babel">
    function Sample(props)
    {       return <h1> welcome to stateless components</h1>                    }
    ReactDOM.render(<Sample/>, document.getElementById("root")
</script>
```

Coding Example 2:Usage of props in stateless components

```
<body>
<div id="root"></div>
<script type = "text/babel">
    function Sample(props)
    {
```

```
            return <h1> {props.name}, welcome to stateless components</h1>
        }

        ReactDOM.render(<div><Sample name = "sindhu"/>
            <Sample name = "thaksh"/>
            <Sample name = "shyama"/>
            </div>
        ,document.getElementById("root"))
    </script>
    </body>
```

Output:



What if we change the rendering code as below? Is it possible to access the content of tag/componenet inside the stateless componenet creation code? If yes, how?

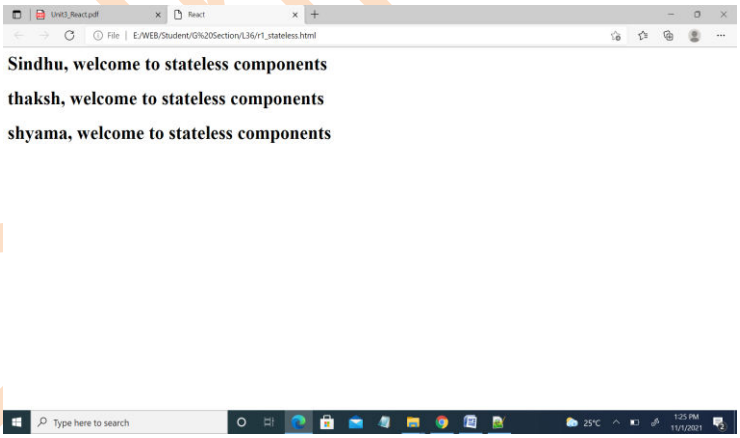
```
ReactDOM.render(<div><Sample>Sindhu</Sample>
            <Sample>thaksh</Sample>
            <Sample>shyama</Sample>
            </div>
        ,document.getElementById("root"))
```

Answer: Possible using props.children

Coding Example 3:Usage of props.children in stateless components

```
<body>
        <div id="root"></div>
        <script type = "text/babel">
                function Sample(props)
                {
                return <h1> {props.children}, welcome to stateless components</h1>
                }
                ReactDOM.render(<div><Sample>Sindhu</Sample>
                        <Sample>thaksh</Sample>
                        <Sample>shyama</Sample>
                        </div>
                ,document.getElementById("root"))
        </script>
</body>
```

Output:



Nesting of stateless Components

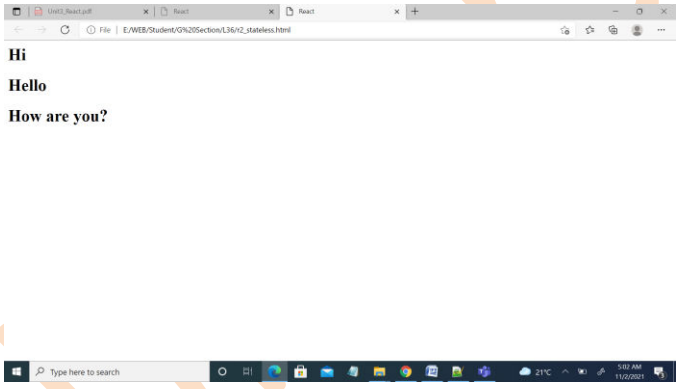
Consider the below code to understand the concept of nesting.

Coding Example 1:

```
<div id="root"></div>
<script type = "text/babel">
```

```
function Sample(props){
        return <Sample1>{props.text}</Sample1> }
function Sample1(props){
        return <h1>{props.children}</h1>  }
ReactDOM.render(<div><Sample     text= "Hi"/>
        <Sample text = "Hello"/>
        <Sample text = "How are you?"/>
        </div>
,document.getElementById("root"))
```

Output:



If we render the component with below code, what changes must be done in the above code? Think!!

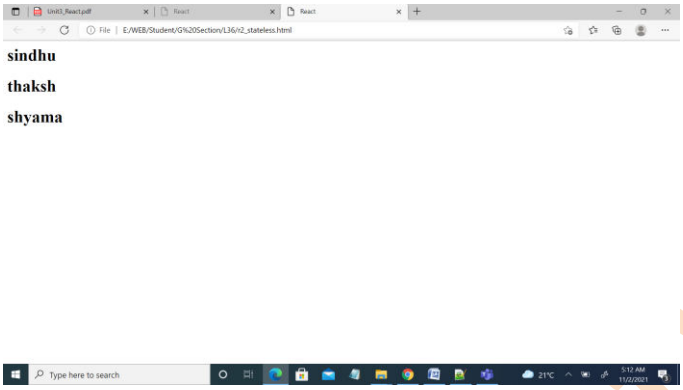
Coding Example 2:

```
ReactDOM.render(<div><Sample>sindhu</Sample>
        <Sample>thaksh</Sample>
        <Sample>shyama</Sample>
        </div>
,document.getElementById("root"))
```

Answer: Only change is as below

```
function Sample(props)
{       return <Sample1>{props.children}</Sample1>      }
```

Output:

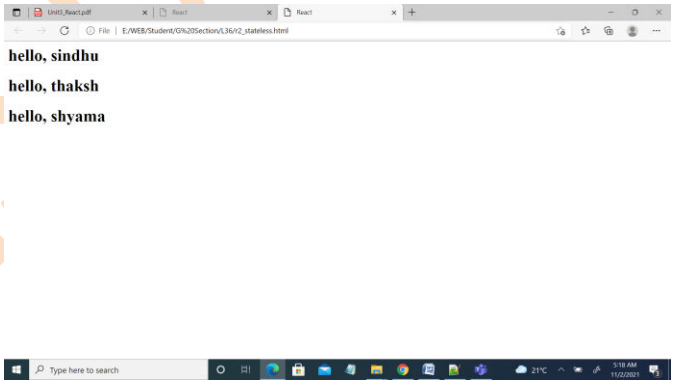


Coding Example 3: Usage of variable within the component and no change in rendering the component

```
function Sample(props)
    {       var greeting = "hello,"
            return <Sample1>{greeting} {props.children}</Sample1>
    }
```

\

Output:



Stateful vs Stateless Components

Stateful Components	Stateless Components
Also known as container or smart components.	Also known as presentational or dumb components.
Have a state	Do not have a state
Can render both props and state	Can render only props
Props and state are rendered like *{this.props.name}* and *{this.state.name}* respectively.	Props are displayed like *{props.name}*
A stateful component is always a *class* component.	A Stateless component can be either a functional or class component.

Refs and Keys in React

Introduction to Refs

Ref provides a way **to access DOM nodes or React elements** created in the render method. It is an **attribute which makes it possible to store a reference to particular DOM nodes or React elements.**

According to React.js documentation some of the best cases for using refs are:

- ➢ managing focus
- ➢ text selection
- ➢ media playback
- ➢ triggering animations
- ➢ integrating with third-party DOM libraries

Usually props are the way for parent components to interact with their children. However, in some cases you might need **to modify a child without re-rendering it with new props**. That's exactly when refs attribute comes to use.

Need of Refs

Consider the below code to fulfil the requirement of clicking on the plus button must increment the value of input text box and clicking on the minus button must decrement the value of text box.

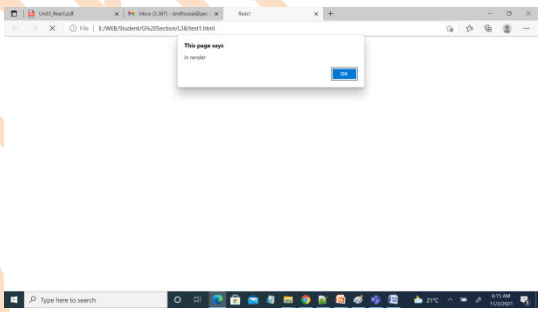
Coding Example 1:

```
<div id="root"></div>
        <script type = "text/babel">
        class My_component extends React.Component
        {       constructor()
                {       super();        this.state = {val:0}            }
                render()
                {       alert("in render");
                        return (<div>
```

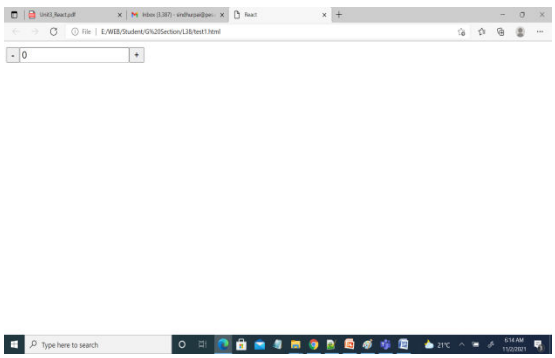
```
                    <button onClick = {this.decrement}>-</button>
                    <input type = "text" value = {this.state.val}/>
                    <button onClick = {this.increment}>+</button>
                    </div>)
                }
                increment=()=>
                {       this.setState({val: this.state.val+1}  )   }
                decrement=()=>
                {       this.setState({val: this.state.val-1})    }
            }
            ReactDOM.render(<My_component/>,
document.getElementById("root"))
        </script>
```

Outputs:

Case 1: Rendered the page on the browser



Case 2:OK is clicked

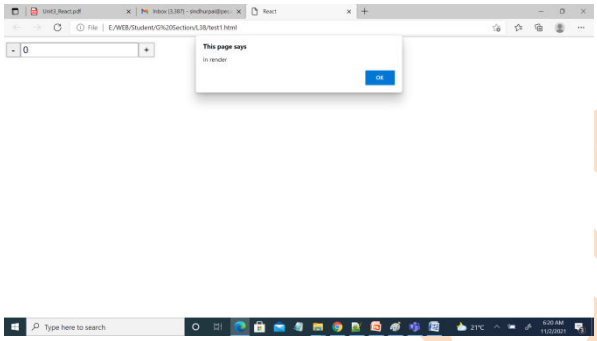


There are two problems in above code.

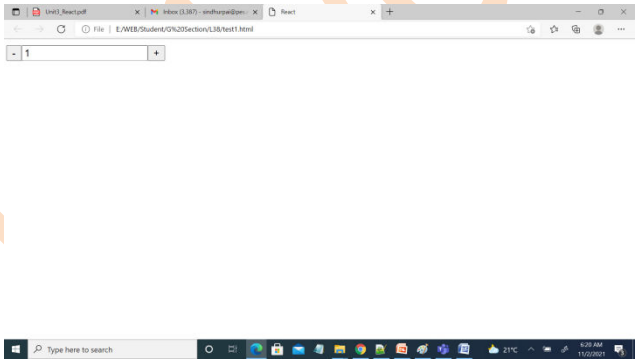
1. When + buton or – button is clicked, the render function is getting called by default as shown in the output below.

2. Unable to edit the input text box

Case 3: + is clicked, observe the mouse pointer

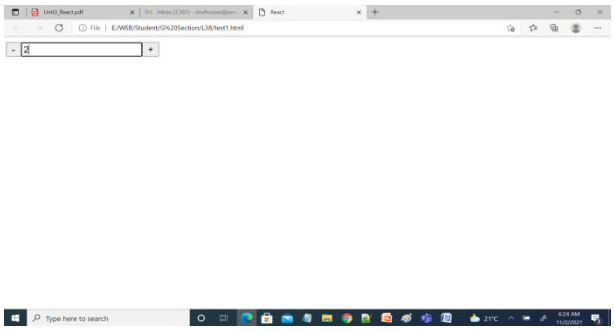


Case 4: Ok is clicked



Note: Same with - button

Case 5:If any key is pressed, input text box no effect



To avoid above problems, we use refs.

Creation and Accessing references

Refs are created using **React.createRef().** Can be assigned to React elements via the **ref attribute**. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

When a ref is passed to an element in render, **a reference to the node** becomes accessible at the **current attribute of the ref.** The value of the ref differs depending on the type of the node:

When used on a HTML element, the ref created in the constructor with React.createRef() receives the underlying DOM element as its current property.

When used on a custom class component, the ref object receives the mounted instance of the component as its current.

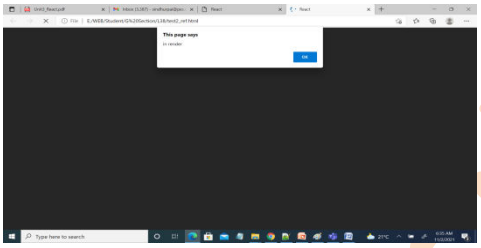
Coding example 2:

```
<div id="root"></div>
        <script type = "text/babel">
                class My_component extends React.Component
                {       constructor()
                        {       super(); this.myref = React.createRef()                 }
                        render()
                        {       alert("in render")
                        return (
                                <input type = "text" ref = {this.myref} />
                                <button onClick = {this.increment}>+</button>
                                </div>
                        )       }
                        increment=()=>
                        {       this.myref.current.value++;                     }
                        decrement=()=>
                        {       this.myref.current.value--;                     }
                }
```

```
        ReactDOM.render(<My_component/>, document.getElementById("root"))
    </script>
```

Outputs:

Case 1: When the page is loaded



Case 2: When OK is clicked

Case 3: + button is clicked, render not called

Case 4: Again + is clicked, render not called

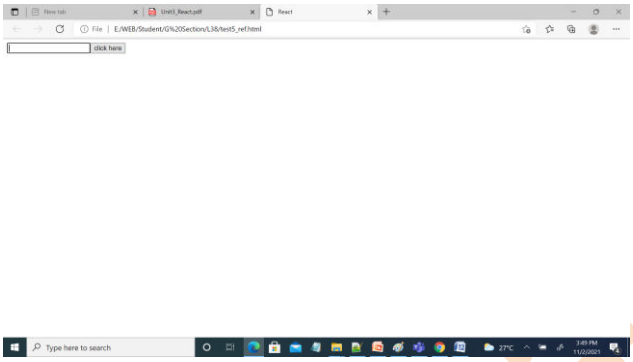
Coding Example 3: Autofocus the input field by adding ref to a DOM element

```
<body>
<div id="root"></div>
<script type = "text/babel">
        class CustomInput extends React.Component
        {       constructor()
                {       super();        this.textref = React.createRef()
                }
                focusTextInput=()=>
                {           this.textref.current.focus()        }
                render()
                {       return (<div>
                                <input type = "text" ref = {this.textref} />
                                <input type = "button" value = "submit" onClick =
                                {this.focusTextInput}/>
                                </div> )
                }
        }
        ReactDOM.render(<CustomInput />,document.getElementById("root"))
</script>
</body>
```

Output:

Case 1: When the page is loaded

Case 2: After clicking on click here button



Think about having no button. When the page is loaded, input box must automatically get focused.

Coding Example 4:

```
<body>
<div id="root"></div>
<script type = "text/babel">
        class CustomInput extends React.Component
        {       constructor()
                {       super(); this.textref = React.createRef()       }
                componentDidMount()
                 //Removed focusInoutText function. Added the functionality in one of the life
cycle method
                {       this.textref.current.focus()                    }
                render()
                {       return (
                                <div>
                                <input type = "text" ref = {this.textref} />
                                </div> )
                }
        }
        ReactDOM.render(<CustomInput />,document.getElementById("root"))
</script>
</body>
```

Few points to think:

- Can we have the ref set for component rather than the DOM Element?
- Can you create more than references for the same element?

Callback refs

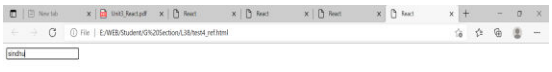
React also supports another way to set refs called "callback refs", which gives more fine-grain control over when refs are set and unset. Instead of passing a ref attribute created by createRef(), you pass a function. **The function receives the React component instance or HTML DOM element as its argument,** which can be stored and accessed elsewhere.

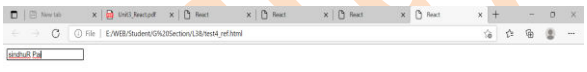
Coding Example 5: Consider the input box. As and when the user types into it, the content of input box is displayed on the page. If the user presses shift key, content has to be displayed in red color

```
<body>
    <div id="root"></div>
    <script type = "text/babel">
        var txt;
        class My_component extends React.Component
        {   constructor()
            {  super(); this.myref = (ele) => {this.setref = ele}// callback ref    }
            render()
            { return(<div>
                    <input type = "text" onKeyPress = {this.show} />
                    <h1 ref = {this.myref}></h1>
                    </div>
            ) }
            show=(e)=>
            {   txt = e.key
                if(e.shiftKey){
                  this.setref.innerHTML += '<span style =
```

```
                    "color:red"}>'+txt+'</span>'   }
                    else {   this.setref.innerHTML += txt   } // current not available
                }
            }
            ReactDOM.render(<My_component/>, document.getElementById("root"))
        </script>
</body>
```

Output:





Introduction to Keys

A key is a unique identifier which helps to identify which items have changed, added, or removed. Useful when we dynamically created components or when users alter the lists. The best way to pick a key is to choose a string that uniquely identifies the items in the list. Keys used within arrays should be unique among their siblings. However, they **don't need to be globally unique.** Also helps in efficiently updating the DOM.

Coding example 6: Consider an array containing n elements in it. Display these elements using n bullet items in an unordered list

```
<script type = "text/babel">
        const arr = ["book1","book2","book3", ","book4"]
```

```
function Booklist(props)
        {       return (<ul> <li >{props.books[0]}</li>
                             <li >{props.books[1]}</li>
                             <li >{props.books[2]}</li>
                             <li >{props.books[3]}</li>
                        </ul>)
        }
    ReactDOM.render(<Booklist books = {arr}/>,document.getElementById("root"))
</script>
```

Observation: As and when the number of elements changes in the array, the code runs into trouble. Refer to the below code to have this dynamism.

Coding example 7:

```
<script type = "text/babel">
    function Booklist(props)
    {
        const book_lists= props.books
        //console.log(book_lists)
        const b = book_lists.map((book,index) => <li key = {index} >{book}</li>)
        return <ul>{b}</ul>
    }
    ReactDOM.render(<Booklist books = {arr}/>,document.getElementById("root"))
</script>
```

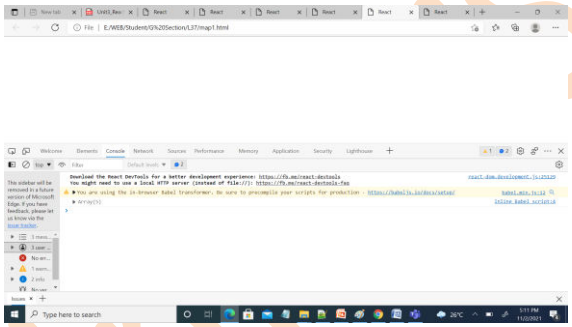
Usage of map

React really simplifies the rendering of lists inside the JSX by supporting the Javascript .map() method. The .map() method in Javascript **iterates through the parent array and calls a function on every element of that array**. **Then it creates a new array with transformed values. It doesn't change the parent array.**

Coding example 8:

```
<body>
        <div id="root"></div>
        <script type = "text/babel">
                const numbers = [1, 2, 3, 4, 5];
                const doubled = numbers.map((number) => number * 2);
                console.log(doubled); //[2, 4, 6, 8, 10]
        </script>
 </body>
```

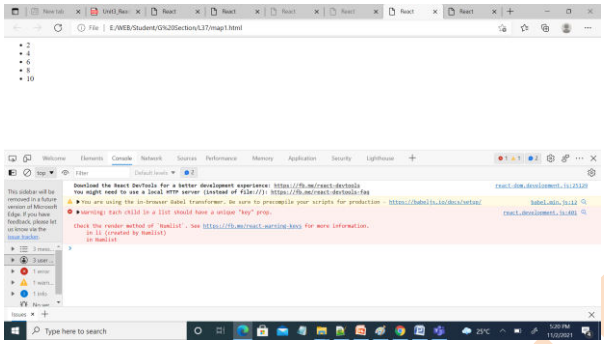
Output:



Coding example 9: Rendering the doubled numbers on the browser

```
<body>
        <div id="root"></div>
        <script type = "text/babel">
        function Numlist()
        {
                const numbers = [1, 2, 3, 4, 5];
                const doubled = numbers.map((number) => <li>{number * 2}</li>);
                return <ul>{doubled}</ul>
        }
        ReactDOM.render(<Numlist />,document.getElementById("root"))
        </script>
 </body>
```

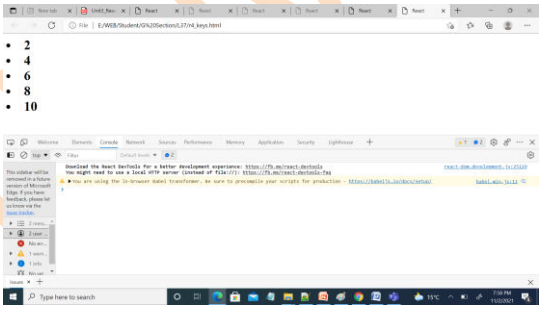
Output:



Note: Please observe the warning. To avoid warning, refer to the below code

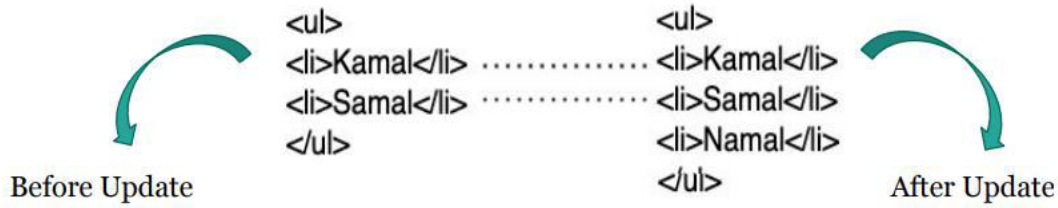
Coding example 10:

The below line of statement, **const doubled = numbers.map((number) => {number * 2});** must be changed to **const doubled = props.numbers.map((number) => <li key = {number.toString()}>{number * 2});**



Importance of keys in react

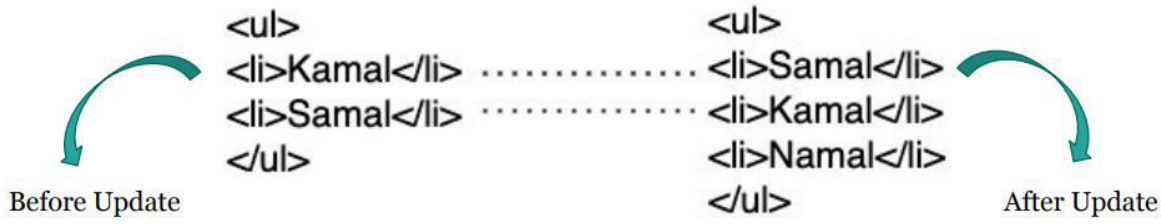
Consider the scenario of updating a list.



In the above case, React will verify that the first and second elements have not been

changed and adds only the last element to the list.

Now, consider a change in the updated list.



In this case, React cannot identify that "Kamal" and "Namal" have not been changed. Therefore, it will update three elements instead of one which will lead to a waste of performance. To solve this, keys are used.

Try writing the code to update the given list using react keys

Event Handling in React

Introduction

Events make the **web app interactive and responsive** to the user. Most of the time, you don't have static UIs; you need to build elements that are smart enough to respond to user actions. So far in the examples we've seen, we brought in a certain level of user interaction such as performing some action on a button click. By now, you would have realized that handling events with React elements is very similar to handling events on DOM elements.

React Event Handling vs DOM Event handling

1. With JSX in ReactJs, you pass a function as event handler and in DOM element we pass function as string

```
// event handling in ReactJs element
<input id="inp" name="name" onChange={onChangeName} />
```

```
// event handling in DOM element
<input id="inp" name="name" onchange="onChangeName()"/>
```

2. In DOM elements the event name is in lowercase while in ReactJs it is in camalCase. List of events are as follows.

Mouse

- onClick
- onContextMenu
- onDoubleClick
- onMouseDown
- onMouseEnter
- onMouseLeave
- onMouseMove
- onMouseOut
- onMouseOver
- onMouseUp

Image

- onLoad
- onError

Selection

- onSelect

Form

- onChange
- onInput
- onSubmit

Focus

- onFocus
- onBlur

Keyboard

- onKeyDown
- onKeyPress
- onKeyUp

UI

- onScroll

3. Cannot return false to prevent default behavior in React. Must call preventDefault explicitly.

HTML

```
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

React

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
      }
  return (<form onSubmit={handleSubmit}>
   <button type="submit">Submit</button>
   </form>);
        }
```

Event Registration

It tells the browser that a particular function should be called whenever a definite event occurs i.e whenever an event is triggered, the function which is bound to that event should be called. Essentially, it allows you to add an event handler for a specified event. This can be used to bind to any event, such as keypress , mouseover or mouseout . Since class methods are not bound by default, it's necessary to bind functions to the class instance so that the this keyword would not return "undefined".

Synthetic Event Objects

React event handling system is known as Synthetic Events. The event object passed to the event handlers are SynthticEvent Objects. It is a wrapper around the DOMEvent object. The event handlers are registered at the time of rendering. Whenever you call an event handler within ReactJS, they are passed an instance of SyntheticEvent. A SyntheticEvent event has all of its usual **properties and methods**. These include its **type, target, mouse coordinates,** and so on. React defines these synthetic events according to the W3C spec to take care of **cross-browser compatibility.**

SyntheticEvent that wraps a **MouseEvent will have access to mouse-specific properties** such as the following:

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

A SyntheticEvent that wraps a **KeyboardEvent will have access to keyboard-related properties** such as the following:

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

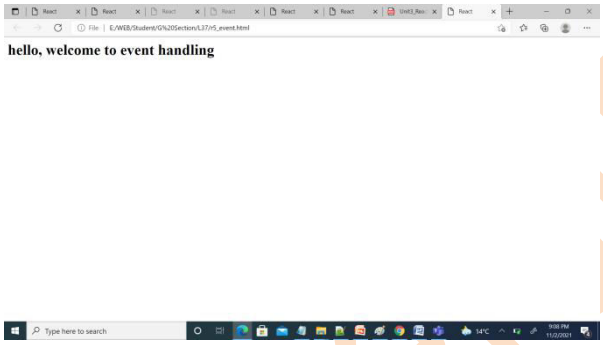
Coding example 1: Simple code to demo event handling

```
<body>
        <div id="root"></div>
        <script type = "text/babel">
            class NewOne extends React.Component     {
                constructor()
                {       super();
                        this.state = {content:"hello, welcome to event handling"}
                }
                render()
                {return <h1 onClick = {this.fun1}>{this.state.content}</h1>  }
                fun1=()=>
                {       this.setState({content:"new text"})             }
```

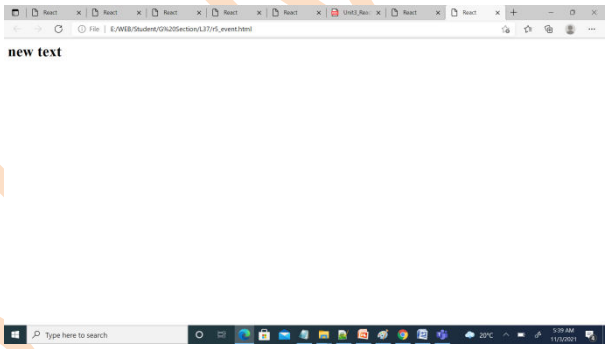
```
        }
        ReactDOM.render(<NewOne/>,document.getElementById("root"))
    </script>
<body>
```

Output:

Case 1: When page is loaded



Case 2: When text is clicked



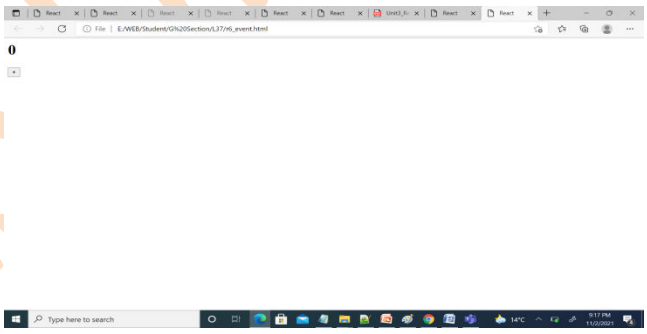
- **Coding example 2: Requirement is to click on the + button, the value of counter must be incremented by one**

```
<body>
    <div id="root"></div>
    <script type = "text/babel">
        class NewOne extends React.Component    {
        constructor()
        {  super();              this.state = {counter:0}        }
```

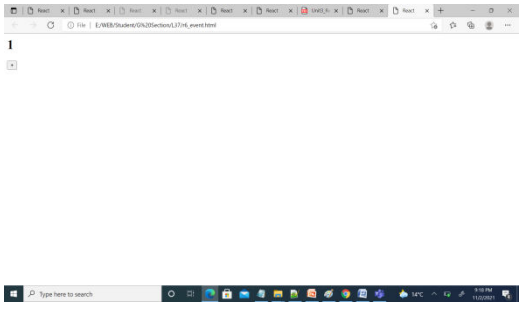
```
            render()        {
                return (<div>
                            <h1 >{this.state.counter}</h1>
                            <button onClick = {this.fun1}>+</button>
                            </div>)

            }
            fun1=()=>
            {       //this.setState(counter:this.state.counter+1})
                    this.setState((prevState) => ({counter:prevState.counter+1}))
            }
        }
        ReactDOM.render(<NewOne/>,document.getElementById("root"))
    </script>
<body>
```

Output:

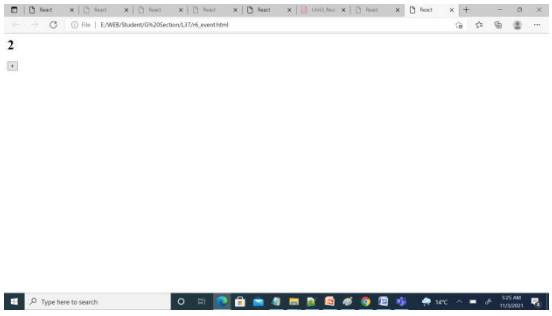
Case 1: when the page is loaded



Case 2: When + button is clicked once

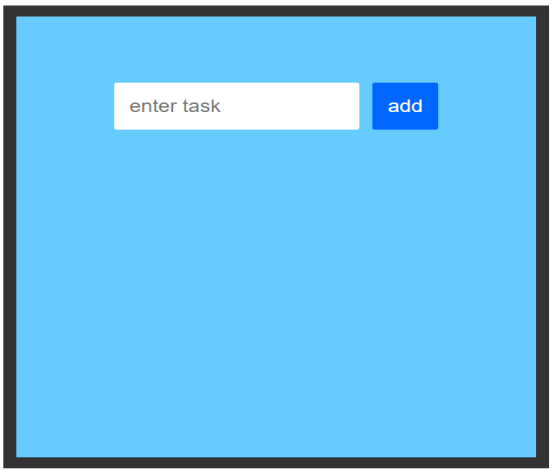


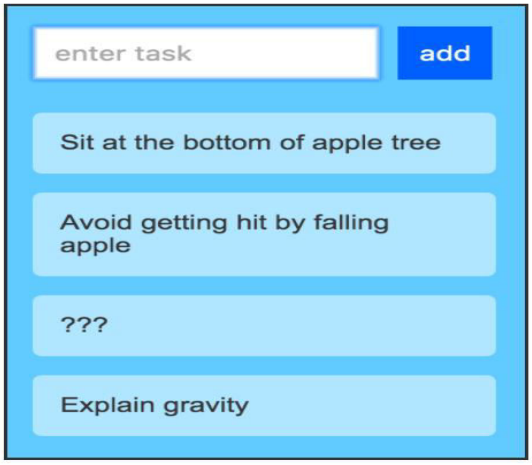
Case 3: When + button is clicked again



Practice Programs

1. Build an awesome todo list as shown below. Type the task in the input box provided and click on add button. This must create an element below the input box with a new background color for this. Clicking on the task directly, must delete the element from the list.





2. Simulate the below to obtain the current date string on the click of a button.

Current Time:

Sat Oct 16 2021 17:10:00 GMT+0530 (India Standard Time)

Get Current Time!

Form Handling in React

Introduction

Forms in React are handled primarily based on the manner in which data is managed. Form handling deals with how to handle the data when the input values are changed or when the form is submitted. Control these changes by adding event handlers to onChange and onSubmit respectively.

HTML Form elements such as <input>, <textarea> and <select> typically maintain their own state and update it based on the user input. The DOM becomes responsible for handling the form data.

Ways to create forms in React

➢ **Uncontrolled Components:** These React Components are traditional HTML form inputs which remember what you typed. **Refs are used get the form values.** Use the **defaultValue** property to specify initial value in React.

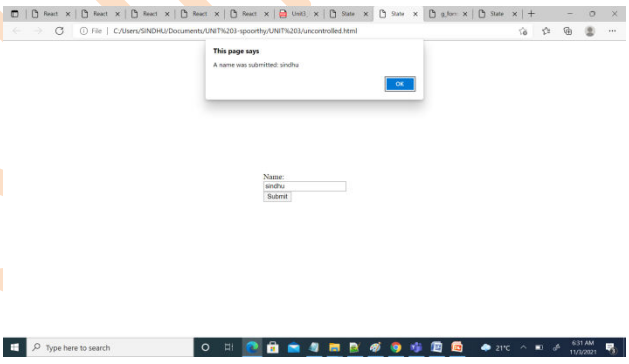
➢ **Controlled Components:** These are React components that render a form and also control what happens in that form on subsequent user input. This means that, as form value changes, the component that renders the form saves the value in its state. The controlled component is a way that you can handle the form input value using the **state** and changing the input value is possible using **setState.**

Coding example 1: Demo of uncontrolled components

```
<body>
  <div id="root"></div>
  <script type="text/babel">
      class UncontrolledForm extends React.Component {
          constructor() {
              super();  this.handleSubmit = this.handleSubmit.bind(this);
```

```
                this.input = React.createRef();
            }
            handleSubmit(event) {
                event.preventDefault();
                alert('A name was submitted: ' + this.input.current.value);
            }
            render() {
                return ( <form onSubmit={this.handleSubmit}>
                Name:<input className="input" type="text" ref={this.input} />
                <input type="submit" value="Submit" />
                </form>       );
                 }
        }
        ReactDOM.render(<UncontrolledForm/>, document.getElementById('root'));
</script>
</body>
```

Output: Input box is filled with a value and submit button is clicked

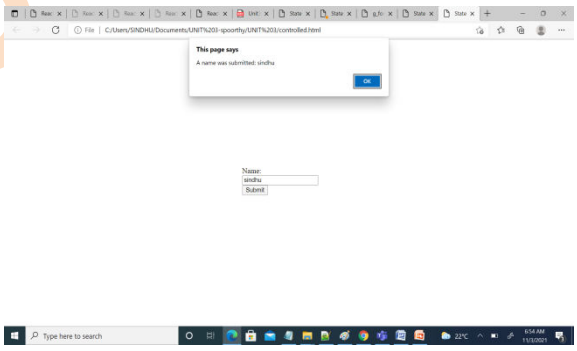


Coding example 2: Demo of controlled components

```
 <body>
   <div id="root"></div>
   <script type="text/babel">
    class ControlledForm extends React.Component {
        constructor(props) { super(props);     this.state = {value: ''};       }
```

```
handleChange=(event)=> {
  //        console.log(this.state.value)
  this.setState({value: event.target.value});        }
handleSubmit=(event)=> {
  event.preventDefault(); alert('A name was submitted: ' + this.state.value); }
render() {
  return (
  <form onSubmit={this.handleSubmit}>
     <label>
     Name:
     <input type="text" value={this.state.value} onChange={this.handleChange} />
     </label>
     <input type="submit" value="Submit" />
  </form> );
 }
}
ReactDOM.render(<ControlledForm/>, document.getElementById('root'));
</script>
</body>
```

Output: Input box is filled with a value and submit button is clicked

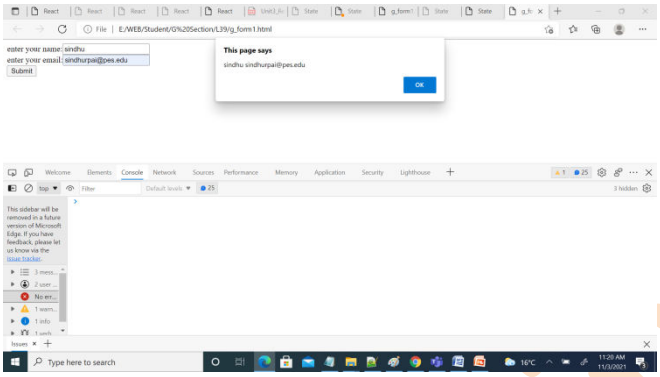


If there is more than one input field, can we have the same handler for onChange event? Explained through below example code

Coding Example 3:

```
<body>
    <div id = "root"> </div>
    <script type = "text/babel">
        class My_form extends React.Component
        {   constructor(){ super();
                this.state = {name:"",email:""  }
            }
            render() {
                return (<form onSubmit = {this.handleSubmit}>
                    <label>enter your name:<input type = "text" name =
"names" onChange = {this.handleChange}/></label><br/>
                    <label>enter your email:<input type = "email" name
= "email" onChange = {this.handleChange}/></label><br/>
                    <input type = "submit" value = "Submit" />
                </form>)                }
            handleChange=(event)=>
            {   var name1 = event.target.name
                var value1 = event.target.value
                if(name1 == "names")
                    this.setState({name:value1})
                if(name1 == "email")
                    this.setState({email:value1})
            }
            handleSubmit=(event)=>
            {   event.preventDefault();
                alert(this.state.name+" "+this.state.email)     }
        }
        ReactDOM.render(<My_form />, document.getElementById("root"))
</script>           </body>
```

Output:



Observation: The above code works absolutely fine. But if there are more number of fields in the form, those many key value pairs must be there in state object. Also, inside the handler, so many times if condition must be used to check for the equality of event.target.name. To avoid this, we use **...this.state.form** as shown in the below code.

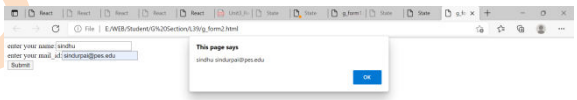
Handling Multiple inputs using React Way

Coding Example 4:Change in the creation of state object and accessing the setState function

```
<body>
<div id = "root"></div>
<script type = "text/babel">
        class My_form extends React.Component{
                constructor(){
                        super();
                        this.state = {form: {names:"", email:""}}
                }
                render(){
                        return (<form onSubmit = {this.handleSubmit}>
                        <label>enter your name:<input type = "text" name = "names"
onChange = {this.handleChange}/></label><br/>
                        <label>enter your mail_id:<input type = "email" name =
```

```
"email" onChange = {this.handleChange}/></label> <br/>
                        <input type = "submit" value = "Submit" />
                        </form> )                    }
                handleChange=(event)=>
                {       var name1 = event.target.name
                        var value1 = event.target.value
                        this.setState({
                                ...this.state.form,
                                form:{
                                        ...this.state.form,
                                        [name1]:[value1]
                                }
                        })                      }
                handleSubmit=(event)=>{
                event.preventDefault()
                alert(this.state.form.names+ " "+this.state.form.email)   }
        }
        ReactDOM.render(<My_form />, document.getElementById("root"))
    </script>
</body>
```

Output:



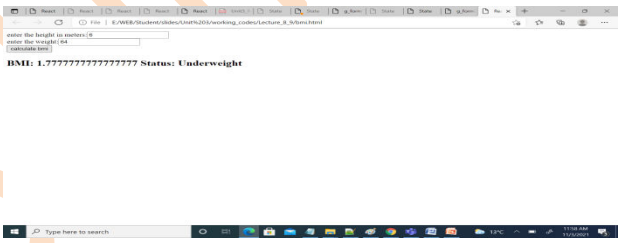
- **Coding example 5:** Calculate the Body Mass Index of a person, given the height in meters and weight in kilograms. Also display appropriate message.

- bmi = weight/(height*height)
  - If bmi < 19, display “underweight”
  - If bmi is between 20 and 24, display “Normal”
  - Else display “overweight”

```
<body>
    <div id="root"></div>
    <script type="text/babel">
        class BMICalc extends React.Component
        {   constructor(){ super()
                this.setHRef=(el)=>{this.heightinput=el}
                this.setWRef=(el)=>{this.weightinput=el}
                this.setStatRef=(el)=>{this.statusoutput=el}
            }
            render()
            {   return(<div>  <form onSubmit = {this.handleSubmit}>
                <label>enter the height in meters:<input type = "text" ref
={this.setHRef}/></label> <br/>
                <label>enter the weight:<input type = "text" ref
={this.setWRef}/></label><br/>
                <input type = "submit" value = "calculate bmi"/>
                </form>
                <h2 ref={this.setStatRef}></h2>
                </div>
            }
            handleSubmit=(event)=>
            {
                event.preventDefault()
                var h = parseFloat(this.heightinput.value)
                var w = parseInt(this.weightinput.value)
                var bmistat;
                var bmi = w/(h*h);
```

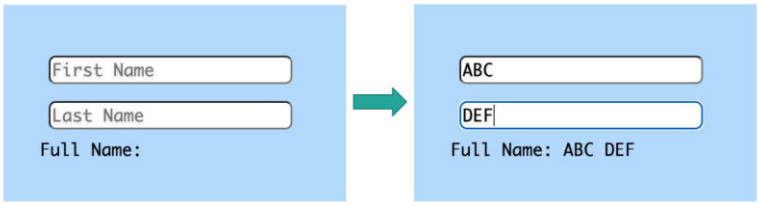
```
                    if(bmi<19)
                        bmistat="Underweight";
                    else if(bmi<25)
                        bmistat="Normal"
                    else
                        bmistat="Overweight"
                    this.statusoutput.innerHTML = "BMI: "+bmi+" Status:
"+bmistat
                }

            }
            ReactDOM.render(<BMICalc/>,document.getElementById("root"))

        </script>

</body>
```

Output:



Practice Problem

1. With the help of a form, accept first and last names from the user separately and display full name. Use one handler function for both the inputs. Additionally, display an error message on entering numerical characters.



Form Handling in React

Introduction

Forms in React are handled primarily based on the manner in which data is managed. Form handling deals with how to handle the data when the input values are changed or when the form is submitted. Control these changes by adding event handlers to onChange and onSubmit respectively.

HTML Form elements such as <input>, <textarea> and <select> typically maintain their own state and update it based on the user input. The DOM becomes responsible for handling the form data.

Ways to create forms in React

➢ **Uncontrolled Components:** These React Components are traditional HTML form inputs which remember what you typed. **Refs are used get the form values.** Use the **defaultValue** property to specify initial value in React.

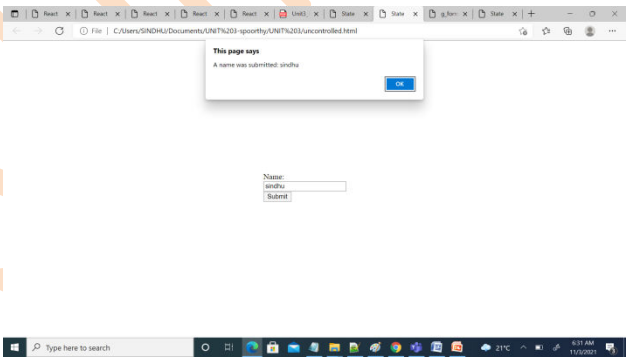
➢ **Controlled Components:** These are React components that render a form and also control what happens in that form on subsequent user input. This means that, as form value changes, the component that renders the form saves the value in its state. The controlled component is a way that you can handle the form input value using the **state** and changing the input value is possible using **setState.**

Coding example 1: Demo of uncontrolled components

```
<body>
  <div id="root"></div>
  <script type="text/babel">
      class UncontrolledForm extends React.Component {
          constructor() {
              super();  this.handleSubmit = this.handleSubmit.bind(this);
```

```
                this.input = React.createRef();
            }
            handleSubmit(event) {
                event.preventDefault();
                alert('A name was submitted: ' + this.input.current.value);
            }
            render() {
                return ( <form onSubmit={this.handleSubmit}>
                Name:<input className="input" type="text" ref={this.input} />
                <input type="submit" value="Submit" />
                </form>       );
                 }
        }
        ReactDOM.render(<UncontrolledForm/>, document.getElementById('root'));
</script>
</body>
```

Output: Input box is filled with a value and submit button is clicked

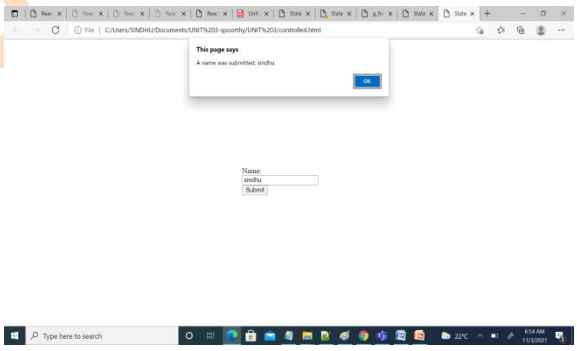


Coding example 2: Demo of controlled components

```
 <body>
   <div id="root"></div>
   <script type="text/babel">
    class ControlledForm extends React.Component {
        constructor(props) { super(props);     this.state = {value: ''};       }
```

```
handleChange=(event)=> {
  //        console.log(this.state.value)
  this.setState({value: event.target.value});        }
handleSubmit=(event)=> {
  event.preventDefault(); alert('A name was submitted: ' + this.state.value); }
render() {
  return (
  <form onSubmit={this.handleSubmit}>
     <label>
     Name:
     <input type="text" value={this.state.value} onChange={this.handleChange} />
     </label>
     <input type="submit" value="Submit" />
  </form> );
 }
}
ReactDOM.render(<ControlledForm/>, document.getElementById('root'));
</script>
</body>
```

Output: Input box is filled with a value and submit button is clicked

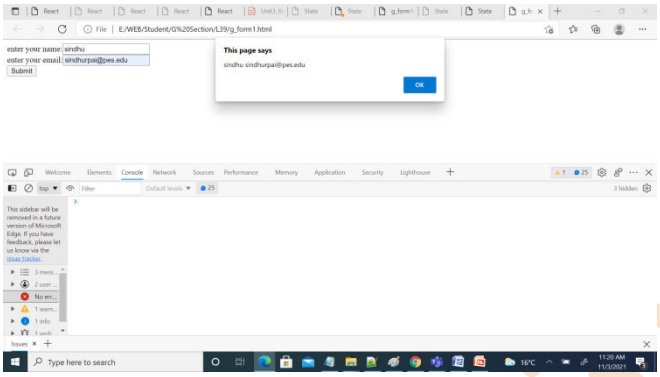


If there is more than one input field, can we have the same handler for onChange event? Explained through below example code

Coding Example 3:

```
<body>
    <div id = "root"> </div>
    <script type = "text/babel">
        class My_form extends React.Component
        {   constructor(){ super();
                this.state = {name:"",email:""  }
            }
            render() {
                return (<form onSubmit = {this.handleSubmit}>
                    <label>enter your name:<input type = "text" name =
"names" onChange = {this.handleChange}/></label><br/>
                    <label>enter your email:<input type = "email" name
= "email" onChange = {this.handleChange}/></label><br/>
                    <input type = "submit" value = "Submit" />
                </form>)                }
            handleChange=(event)=>
            {   var name1 = event.target.name
                var value1 = event.target.value
                if(name1 == "names")
                    this.setState({name:value1})
                if(name1 == "email")
                    this.setState({email:value1})
            }
            handleSubmit=(event)=>
            {   event.preventDefault();
                alert(this.state.name+" "+this.state.email)     }
        }
        ReactDOM.render(<My_form />, document.getElementById("root"))
</script>           </body>
```

Output:



Observation: The above code works absolutely fine. But if there are more number of fields in the form, those many key value pairs must be there in state object. Also, inside the handler, so many times if condition must be used to check for the equality of event.target.name. To avoid this, we use **...this.state.form** as shown in the below code.

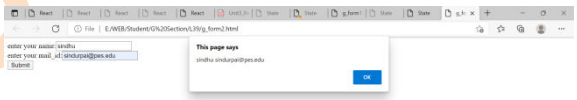
Handling Multiple inputs using React Way

Coding Example 4:Change in the creation of state object and accessing the setState function

```
<body>
<div id = "root"></div>
<script type = "text/babel">
        class My_form extends React.Component{
                constructor(){
                        super();
                        this.state = {form: {names:"", email:""}}
                }
                render(){
                        return (<form onSubmit = {this.handleSubmit}>
                        <label>enter your name:<input type = "text" name = "names"
onChange = {this.handleChange}/></label><br/>
                        <label>enter your mail_id:<input type = "email" name =
```

```
"email" onChange = {this.handleChange}/></label> <br/>
                        <input type = "submit" value = "Submit" />
                        </form> )                    }
                handleChange=(event)=>
                {       var name1 = event.target.name
                        var value1 = event.target.value
                        this.setState({
                                ...this.state.form,
                                form:{
                                        ...this.state.form,
                                        [name1]:[value1]
                                }
                        })                      }
                handleSubmit=(event)=>{
                event.preventDefault()
                alert(this.state.form.names+ " "+this.state.form.email)   }
        }
        ReactDOM.render(<My_form />, document.getElementById("root"))
    </script>
</body>
```

Output:



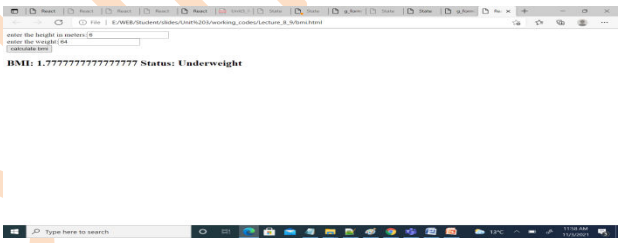
- **Coding example 5:** Calculate the Body Mass Index of a person, given the height in meters and weight in kilograms. Also display appropriate message.

- bmi = weight/(height*height)
  - If bmi < 19, display “underweight”
  - If bmi is between 20 and 24, display “Normal”
  - Else display “overweight”

```
<body>
    <div id="root"></div>
    <script type="text/babel">
        class BMICalc extends React.Component
        {   constructor(){ super()
                this.setHRef=(el)=>{this.heightinput=el}
                this.setWRef=(el)=>{this.weightinput=el}
                this.setStatRef=(el)=>{this.statusoutput=el}
            }
            render()
            {   return(<div>  <form onSubmit = {this.handleSubmit}>
                <label>enter the height in meters:<input type = "text" ref
={this.setHRef}/></label> <br/>
                <label>enter the weight:<input type = "text" ref
={this.setWRef}/></label><br/>
                <input type = "submit" value = "calculate bmi"/>
                </form>
                <h2 ref={this.setStatRef}></h2>
                </div>
            }
            handleSubmit=(event)=>
            {
                event.preventDefault()
                var h = parseFloat(this.heightinput.value)
                var w = parseInt(this.weightinput.value)
                var bmistat;
                var bmi = w/(h*h);
```

```
                    if(bmi<19)
                        bmistat="Underweight";
                    else if(bmi<25)
                        bmistat="Normal"
                    else
                        bmistat="Overweight"
                    this.statusoutput.innerHTML = "BMI: "+bmi+" Status:
"+bmistat
                }

            }
            ReactDOM.render(<BMICalc/>,document.getElementById("root"))

        </script>

</body>
```

Output:



Practice Problem

1. With the help of a form, accept first and last names from the user separately and display full name. Use one handler function for both the inputs. Additionally, display an error message on entering numerical characters.