MODULE 4 PART A

1)HOW TO DECLARE A STATE OBJECT?

React components has a built-in `state` object.

The `state` object is where you store property values that belongs to the component.

When the `state` object changes, the component re-renders.

Creating the `state` Object

The `state` object is initialized in the constructor:

Example:

Specify the `state` object in the constructor method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {brand: "Ford"};
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
}
```

The state object can contain as many properties as you like:

Example:

Specify all the properties your component need:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
}
```

2)Write a program to use and update the state object?

Refer to the state object anywhere in the component by using the this.state.*propertyname* syntax:

Example:

Refer to the state object in the render() method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
```

```
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
      </div>
    );
  }
}
```

Changing the state Object

To change a value in the state object, use the this.setState() method.

When a value in the state object changes, the component will re-render, meaning that the output will change according to the new value(s).

Example:

Add a button with an onClick event that will change the color property:

```
class Car extends React.Component {
 constructor(props) {
 super(props);
 this.state = {
 brand: "Ford",
```

```
 model: "Mustang",
 color: "red",
 year: 1964
 };
 }
 changeColor = () => {
 this.setState({color: "blue"});
 }
 render() {
 return (
 <div>
 <h1>My {this.state.brand}</h1>
 <p>
 It is a {this.state.color}
 {this.state.model}
 from {this.state.year}.
 </p>
 <button
 type="button"
 onClick={this.changeColor}
 >Change color</button>
 </div>
 );
 }
}
```

3)write a program to create an error boundary to handle errors in the error phase?

A JavaScript error in a part of the UI shouldn’t break the whole app. To solve this problem for React users, React 16 introduces a new concept of an “error boundary”.

Error boundaries are React components that **catch JavaScript errors anywhere in their child component**

tree, log those errors, and display a fallback UI instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

Note

Error boundaries do **not** catch errors for:

- Event handlers (learn more)
- Asynchronous code (e.g. setTimeout or requestAnimationFrame callbacks)
- Server side rendering
- Errors thrown in the error boundary itself (rather than its children)

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods static getDerivedStateFromError() or componentDidCatch(). Use static getDerivedStateFromError() to render a fallback UI after an error has been thrown. Use componentDidCatch() to log error information.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null, errorInfo: null };
  }

  componentDidCatch(error, errorInfo) {
```

```
    // Catch errors in any components below and re-
render with error message
    this.setState({
      error: error,
      errorInfo: errorInfo
    })
    // You can also log error messages to an error
reporting service here
  }

  render() {
    if (this.state.errorInfo) {
      // Error path
      return (
        <div>
          <h2>Something went wrong.</h2>
          <details style={{ whiteSpace: 'pre-wrap'
}}>
            {this.state.error &&
this.state.error.toString()}
            <br />
            {this.state.errorInfo.componentStack}
          </details>
        </div>
      );
    }
```

```
    // Normally, just render children
    return this.props.children;
  }
}

class BuggyCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { counter: 0 };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(({counter}) => ({
      counter: counter + 1
    }));
  }

  render() {
    if (this.state.counter === 5) {
      // Simulate a JS error
      throw new Error('I crashed!');
    }
```

```
    return <h1
onClick={this.handleClick}>{this.state.counter}</h1
>;
  }
}


function App() {
  return (
    <div>
      <p>
        <b>
          This is an example of error boundaries in
React 16.
          <br /><br />
          Click on the numbers to increase the
counters.
          <br />
          The counter is programmed to throw when
it reaches 5. This simulates a JavaScript error in
a component.
        </b>
      </p>
      <hr />
      <ErrorBoundary>
        <p>These two counters are inside the same
error boundary. If one crashes, the error boundary
will replace both of them.</p>
```

```
          <BuggyCounter />
          <BuggyCounter />
        </ErrorBoundary>
        <hr />
        <p>These two counters are each inside of
their own error boundary. So if one crashes, the
other is not affected.</p>
        <ErrorBoundary><BuggyCounter
/></ErrorBoundary>
        <ErrorBoundary><BuggyCounter
/></ErrorBoundary>
      </div>
    );
}

const root =
ReactDOM.createRoot(document.getElementById('root')
);
root.render(<App />);
```

4)write a program to prevent rerendering?

If you're using a React class component you can use the shouldComponentUpdate method or a React.PureComponent class extension to prevent a component from re-rendering. But, is there an option to prevent re-rendering with functional components?

The answer is yes! Use `React.memo()` to prevent re-rendering on React function components.

The answer: use React.memo()

```
React.memo(YourComponent);
```

Before I jump into the `React.memo()` code example, let's cover a couple basic things about React functional components.

What is a React functional component and how to create them

React functional components are React components built in a function rather than class.

```
// Style 1
function NormalFunctionComponent() {
 return <h1>Greetings earlthing!</h1>
}

// Style 2
const FunctionComponentWithCurlies = () => {
 return <h1>Greetings earlthing!</h1>
}

// Style 3
const FunctionComponentWithParanthesis = () => (
 <h1>Greetings earlthing!</h1>
);
```

As you can see above, there are 3 styles of creating a React functional component.

The one that really stands out the most is style #3. In that style, you can't add any logic into the function.

React functional components are always the render lifecycle method

React functional components do not have lifecycle methods.

Let's study the code below.

```
const Greeting = props => {
 console.log('Greeting Comp render');
 return <h1>Hi {props.name}!</h1>
};

function App() {
 const [counter, setCounter] = React.useState(0);

 // Update state variable `counter`
 // every 2 seconds.
 React.useEffect(() => {
 setInterval(() => {
 setCounter(counter + 1);
 }, 2000);
 }, []);

 console.log('App render')
 return <Greeting name="Ruben" />
}
```

Since React functional components don't have access to `shouldComponentUpdate()`, and it's not a React pure component class, I'll have to use a new strategy to achieve a pure like function behavior.

Memoization a React functional component with React.memo()

To optimize, and prevent multiple React renders, I'm going to use another React tool called `React.memo()`.

```
const Greeting = React.memo(props => {
 console.log("Greeting Comp render");
 return <h1>Hi {props.name}!</h1>;
});
```

All I need to do in the Greeting component is wrap it another function called `React.memo()`.

5)WHAT ARE THE DIFFERENT WAYS TO STYLE THE REACT COMPONENT?

React is a JavaScript library for building user interfaces. React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

There are about eight different ways to styling React Js components, there names and explanations of some of them are mentioned below.

1. Inline CSS
2. Normal CSS
3. CSS in JS
4. Styled Components
5. CSS module
6. Sass & SCSS
7. Less
8. Stylable

Inline CSS: In inline styling basically we create objects of style. And render it inside the components in style attribute using the React

technique to incorporate JavaScript variable inside the JSX (Using '{ }')

index.js:

-

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './App'

ReactDOM.render(<App />, document.querySelector('#root'))
```

App.js

-

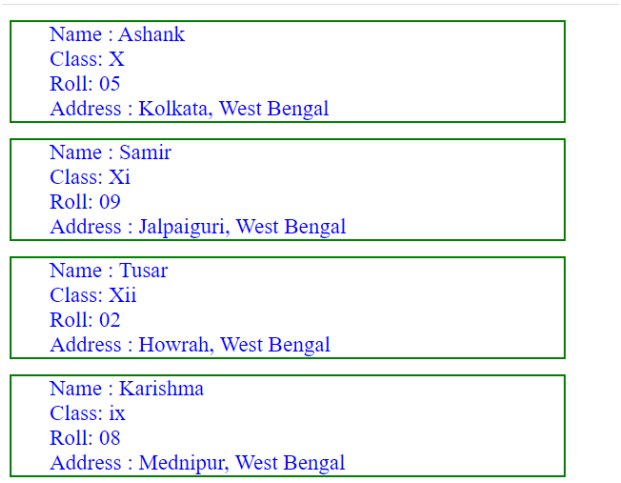
```
import React, { Component } from 'react'
import StudentList from './StudentList'

const App = () => {
  return (
    <div>
      <StudentList
        name='Ashank'
        classNo='X'
        roll='05'
        addr='Kolkata, West Bengal'
      />
      <StudentList
        name='Samir'
        classNo='Xi'
        roll='09'
        addr='Jalpaiguri, West Bengal'
      />
      <StudentList
        name='Tusar'
        classNo='Xii'
        roll='02'
        addr='Howrah, West Bengal'
      />
      <StudentList
        name='Karishma'
        classNo='ix'
        roll='08'
        addr='Mednipur, West Bengal'
      />
    </div>
  )
}
```

```
export default App
```

StudentList.js: It is a studentList component rendered by the App component. It displays details of each student.

-

```
import React, { Component } from 'react'

class StudentList extends Component{
  render(){
    const {name, classNo, roll, addr} = this.props
    const ulStyle = {border: '2px solid green', width:'40%',
listStyleType:'none'}
    const liStyle = {color : 'blue', fontSize:'23px'}
    return(
      <ul style={ulStyle}>
        <li style={liStyle}>Name : {name}</li>
        <li style={liStyle}>Class: {classNo}</li>
        <li style={liStyle}>Roll: {roll}</li>
        <li style={liStyle}>Address : {addr}</li>
      </ul>
    )
  }
}

export default StudentList
```



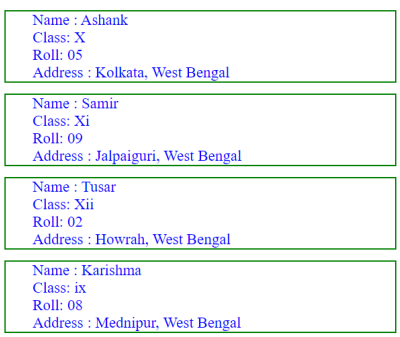
Normal CSS: In the external CSS styling technique, we basically create an external CSS file for each component and do the required styling of classes. and use those class names inside the component. It is a convention that name of the external CSS file same as the name

of the component with '.css' extension. It is better if the name of the classes used, follow the format **'componentName-context'** (here context signifies where we use this classname). For example, if we style the header of a component called 'Box', a better classname should style this element should be **'Box-header'.**

INDEX.JS,APP.JS STUDENTLIST.JS ARE SAME

StudentList.CSS: Since there is no place to add CSS code externally, Here I add a screenshot of the CSS code.

```
.StudentList{
    border: 2px solid green;
    width: 40%:
    list-style-type: none;
    }

.StudentList-details{
    color: blue;
    font-size: 23px;
    }
```



CSS in JS: The'react-jss' integrates JSS with react app to style components. It helps to write CSS with Javascript and allows us to describe styles in a more descriptive way. It uses javascript objects to describe styles in a declarative way using 'createUseStyles' method of react-jss and incorporate those styles in functional components using className attribute

npm install react-jss

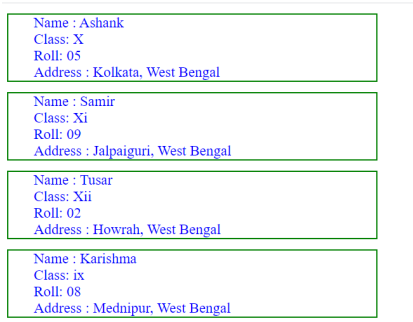
REMAINING SAME

StudentList.js: It is a studentList component rendered by the App component. It displays the details of each student.

-

```
import React, { Component } from 'react'
import {createUseStyles} from 'react-jss'

const styles = createUseStyles({
  student : {
    border : '2px solid green',
    width: '40%',
    listStyleType:'none'
  },

  studentDetails : {
    color : 'blue',
    fontSize : '23px'
  }
})

const StudentList = (props) => {
  const classes = styles()
  const {name, classNo, roll, addr} = props
  return(
    <ul className={classes.student}>
      <li className={classes.studentDetails}>Name : {name}</li>
      <li className={classes.studentDetails}>Class: {classNo}</li>
      <li className={classes.studentDetails}>Roll: {roll}</li>
      <li className={classes.studentDetails}>Address : {addr}</li>
    </ul>
  )
}

export default StudentList
```

-



Styled Components: The styled-components allows us to style the CSS under the variable created in JavaScript. style components is a third party package using which we can create a component as a JavaScript variable that is already styled with CSS code and used that styled component in our main component. styled-components allow us to create custom reusable components which can be less of a hassle to maintain.

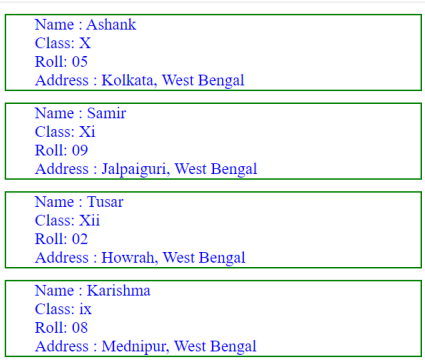
```
npm install --save styled-components
```

StudentList.js: It is a studentList component rendered by the App component. It displays details of each student.

-

```
import React, { Component } from 'react'
import styled from 'styled-components'

//styled component Li
const Li = styled.li`
  color : blue;
  font-size : 23px
`

//Styled component Ul
const Ul = styled.ul`
  border : 2px solid green;
  width: 40%;
  list-style-type:none
```

```
`


const StudentList = (props) => {
  const {name, classNo, roll, addr} = props
  return(
    <Ul>
      <Li>Name : {name}</Li>
      <Li>Class: {classNo}</Li>
      <Li>Roll: {roll}</Li>
      <Li>Address : {addr}</Li>
    </Ul>
  )
}

export default StudentList
```



6) WRITE DOWN THE PROGRAM TO CREATE A SWITCHING COMPONENT FOR DISPLAYING DIFFERENT PAGES?

React Evergreen is a popular front-end library with a set of React components for building beautiful products as this library is flexible, sensible defaults, and User friendly. Switch Component allows the user to toggle the state of a single setting on or off. We can use the following approach in ReactJS to use the Evergreen Switch Component.

Switch Props:

- **id:** It is used to denote the *id* attribute of the radio.
- **name:** It is used to define the name attribute of the radio.
- **value:** It is used to denote the value attribute of the radio.
- **height:** It is used to define the height of the switch.
- **checked:** The switch is checked when this is set to true.
- **onChange:** It is a function that is called when the state changes.
- **disabled:** The switch is disabled when this is set to true.
- **isInvalid:** The switch is invalid when this is set to true.
- **appearance:** It is used for the appearance of the checkbox.

- **hasCheckIcon:** The switch has a check icon when this is set to true.
- **defaultChecked:** The switch is true by default when this is set to true.

App.js

```
import React from "react";
import {
  BrowserRouter as Router,
  Route,
  Switch
} from "react-router-dom";
import Products from "./pages"
import Product from "./pages/product"

const App = () => {
  return (
    <Router>
      <Switch>
        <Route exact path="/" ><Products /></Route>
        <Route path="/product"><Product /></Route>
      </Switch>
    </Router>
  );
}

export default App;

export default Switch;
```

file index.js

```
import React from "react";
import {Link} from "react-router-dom";
const Products = () => {
  return (
    <div>
      <h3>Products</h3>
      <Link to="/product" >Go to product</Link>
    </div>
  );
};
export default Products;
```

file product.js

```
import React from "react";
const Product = () => {
  return (
    <div>
```

```
      <h3>Product !!!!!!</h3>
    </div>
  );
}
export default Product;
```

7)HOW TO RERENDER THE VIEW WHEN THE BROWSER IS RESIZED?

Rerender the View on Browser Resize with React

We can use the useLayoutEffect to add the event listener that runs when the window resizes.

And in the window resize event handler, we can run our code to change a state to make the view rerender.

For instance, we can write:

```
import React, { useLayoutEffect, useState } from "react";

const useWindowSize = () => {
  const [size, setSize] = useState([0, 0]);
  useLayoutEffect(() => {
    const updateSize = () => {
      setSize([window.innerWidth, window.innerHeight]);
    };
    window.addEventListener("resize", updateSize);
    updateSize();
    return () => window.removeEventListener("resize", updateSize);
  }, []);
  return size;
};

export default function App() {
  const [width, height] = useWindowSize();
  return (
    <span>
      Window size: {width} x {height}
    </span>
  );
}
```

We create the useWindowSize hook that has the size state.

In the useLayoutEffect hook callback, we create the updateSize function that calls setSize to set thje size state to an array with window.innerWidth and window.innerHeight , which has the width and height of the browser window respectively.

Then we call window.addEventListener with 'resize' to set updateSize as the window resize event listener.

Also, we return a function that calls window.removeEventListener to clear the resize event listener when the component unmounts.

Finally, we return the size at the end of the hook function.

Then in App , we destructure the width and height returned from the useWindowSize hook.

And then we render it in the span.

Now when the window resizes, we see its size change.

Conclusion

We can use the useLayoutEffect to add the event listener that runs when the window resizes.

And in the window resize event handler, we can run our code to change a state to make the view rerender.

8)WRITE DOWN THE PROGRAM TO PERFORM AUTOMATIC REDIRECT AFTER LOGIN?

Complete Source Code

index.js

index.js will be the entry file of our web application. Here, we will mount the root component to an element, i.e., a with an id of root.

```
import React from "react";
import ReactDOM from "react-dom";

import App from "./App";

const rootElement = document.getElementById("root");

ReactDOM.render(<App />, rootElement);
```

jsx

App.js

In the App.js file, we have defined the main or the root component, i.e., the <App /> component. We will be wrapping all the child components in the <Provider /> component from the react-redux library to make the global redux store available throughout the application.

```
import React from "react";
import { BrowserRouter as Router, Switch, Route } from "react-router-dom";
import "./app.css";
import { Provider } from "react-redux";
import { applyMiddleware } from "redux";
```

```
import reducer from "./reducer";
import { createStore } from "redux";

import NavBar from "./components/Nav";
import { Typography, Divider } from "@material-ui/core";

import AuthRoute from "./components/AuthRoute";

import HomePage from "./pages/HomePage";
import LoginPage from "./pages/Login";

import { appMiddleware } from "./middlewares/app";
import { apiMiddleware } from "./middlewares/core";
import MyAccount from "./pages/MyAccount";

const createStoreWithMiddleware = applyMiddleware(
  appMiddleware,
  apiMiddleware
)(createStore);

const store = createStoreWithMiddleware(reducer);

const IndexPage = () => (
  <>
    <Typography variant="h3">Welcome to the App</Typography>
    <Divider style={{ marginTop: 10, marginBottom: 10 }} />
    <Typography variant="h6">Feel free to take a look around</Typography>
  </>
);

export default function App() {
  return (
    <Provider store={store}>
      <Router>
        <NavBar />
        <div className="container">
          <Switch>
            <AuthRoute path="/home" render={HomePage} type="private" />
            <AuthRoute path="/login" type="guest">
              <LoginPage />
            </AuthRoute>
            <AuthRoute path="/my-account" type="private">
              <MyAccount />
            </AuthRoute>
            <Route path="/" render={IndexPage} />
          </Switch>
        </div>
      </Router>
    </Provider>
```

```
  );
}
```

jsx

Notice that we have used the <Router /> and <Switch /> components from react-router-dom for leveraging client-side routing.

reducer.js

```
import { SET_LOADER } from "./actions/ui";
import { API_SUCCESS, API_ERROR } from "./actions/api";
import { LOGOUT } from "./actions/auth";

export default (
  state = {
    isAuthUser: !!localStorage.getItem("user"),
    user: JSON.parse(localStorage.getItem("user")) || {},
    isLoading: false,
    error: null
  },
  action
) => {
  switch (action.type) {
    case API_SUCCESS:
      localStorage.setItem("user", JSON.stringify(action.payload.user));
      return { ...state, isAuthUser: true, user: action.payload.user };
    case API_ERROR:
      return { ...state, error: action.payload };
    case SET_LOADER:
      return { ...state, isLoading: action.payload };
    case LOGOUT:
      localStorage.removeItem("user");
      return { ...state, isAuthUser: false, user: {} };
    default:
      return state;
  }
};
```

js

MIDDLEWARES

We'll be creating two types of redux middlewares: an appMiddleware and a coreMiddleware. The appMiddeware will be responsible for handling the API requests. In this case, we pass the relevant data for the API request through the LOGIN action, and in the coreMiddleware, we catch the API_REQUEST action and make the network request using the axios HTTP library.

app.js

```
import { apiRequest } from "../actions/api";
import { LOGIN } from "../actions/auth";

const SERVER_URL = `https://61m46.sse.codesandbox.io`;

export const appMiddleware = () => next => action => {
  next(action);
  switch (action.type) {
    case LOGIN: {
      next(
        apiRequest({
          url: `${SERVER_URL}/login`,
          method: "POST",
          data: action.payload
        })
      );
      break;
    }
    default:
      break;
  }


};
```

js

core.js

```
import axios from "axios";
import { API_REQUEST, apiError, apiSuccess } from "../actions/api";
import { setLoader } from "../actions/ui";

export const apiMiddleware = ({ dispatch }) => next => action => {
  next(action);

  if (action.type === API_REQUEST) {
    dispatch(setLoader(true));
    const { url, method, data } = action.meta;
    axios({
      method,
      url,
      data
    })
      .then(({ data }) => dispatch(apiSuccess({ response: data })))
      .catch(error => {
        console.log(error);
        dispatch(apiError({ error: error.response.data }));
      });
  }
```

```
};
```

js

COMPONENTS

AuthRoute.js
The <AuthRoute /> component is a higher-order component that wraps the <Route /> component of react-router to keep the routes specific to our application as private or public.

```
import React from "react";
import { connect } from "react-redux";
import { Redirect, Route } from "react-router";

const AuthRoute = props => {
  const { isAuthUser, type } = props;
  if (type === "guest" && isAuthUser) return <Redirect to="/home" />;
  else if (type === "private" && !isAuthUser) return <Redirect to="/" />;

  return <Route {...props} />;
};

const mapStateToProps = ({ isAuthUser }) => ({
  isAuthUser
});



export default connect(mapStateToProps)(AuthRoute);
```

jsx

Nav.js
In the <Navbar /> component, we are merely creating the navigation menu for our application. Notice that I have used the <AppBar /> component from material-ui to give it a native look.

```
import React, { Component } from "react";
import { Link } from "react-router-dom";
import { AppBar, Toolbar, Button, Typography } from "@material-ui/core";
import { connect } from "react-redux";

import { logout } from "../actions/auth";

class NavBar extends Component {
  render() {
    return (
      <AppBar position="static" style={{ display: "flex" }}>
        <Toolbar>
          <Typography variant="h6">My App</Typography>
          <div style={{ marginLeft: "auto" }}>
            {this.props.isAuthUser ? (
```

```
              <>
                <Link to="/home">
                  <Button color="inherit">Home</Button>
                </Link>
                <Link to="/my-account">
                  <Button color="inherit">My Account</Button>
                </Link>
                <Button color="inherit" onClick={this.props.logout}>
                  Logout
                </Button>
              </>
            ) : (
              <Link to="/login">
                <Button color="inherit">Login</Button>
              </Link>
            )}
          </div>
        </Toolbar>
      </AppBar>
    );
  }
}

export default connect(({ isAuthUser }) => ({ isAuthUser }), { logout })(
  NavBar

);
```

jsx

PAGES

Login.js

In the Login.js file, we are creating the page component that displays the login form. We have used the modern React hooks to leverage state in a functional component. This allows us to make the code more precise and easier to maintain.

```
import React, { useState } from "react";
import { TextField, Typography, Button } from "@material-ui/core";
import { connect } from "react-redux";
import { login } from "../actions/auth";
import MuiAlert from "@material-ui/lab/Alert";

function Alert(props) {
  return <MuiAlert elevation={6} variant="filled" {...props} />;
}

export default connect(({ isLoading }) => ({ isLoading }), { login
})(props => {
  const [email, setEmail] = useState("");
```

```
const [password, setPassword] = useState("");
const [error, setError] = useState("");

const submitForm = () => {
  if (email === "" || password === "") {
    setError("Fields are required");
    return;
  }
  props.login({ email, password });
};

return (
  <form>
    <Typography variant="h5" style={{ marginBottom: 8 }}>
      Login
    </Typography>
    <TextField
      label="Email"
      variant="outlined"
      fullWidth
      className="form-input"
      value={email}
      onChange={e => setEmail(e.target.value)}
    />
    <TextField
      label="Password"
      variant="outlined"
      fullWidth
      className="form-input"
      type="password"
      value={password}
      onChange={e => setPassword(e.target.value)}
    />
    <Button
      variant="contained"
      color="primary"
      fullWidth
      className="form-input"
      size="large"
      onClick={submitForm}
    >
      Login
    </Button>

    {(props.error || error) && (
      <Alert severity="error" onClick={() => setError(null)}>
        {props.error || error}
      </Alert>
    )}
  </form>
```

```
  );

});
```

jsx

9)WRITE DOWN THE PROGRAM TO CREATE FORM IN REACT?

```
import React, {useState} from 'react';
import ReactDOM from 'react-dom';
import './App.css';

function App() {
const [name , setName] = useState('');
const [age , setAge] = useState('');
const [email , setEmail] = useState('');
const [password , setPassword] = useState('');
const [confPassword , setConfPassword] = useState('');

// function to update state of name with
// value enter by user in form
const handleChange =(e)=>{
setName(e.target.value);
}
// function to update state of age with value
// enter by user in form
const handleAgeChange =(e)=>{
setAge(e.target.value);
}
// function to update state of email with value
// enter by user in form
const handleEmailChange =(e)=>{
setEmail(e.target.value);
```

```
}
// function to update state of password with
// value enter by user in form
const handlePasswordChange =(e)=>{
setPassword(e.target.value);
}
// function to update state of confirm password
// with value enter by user in form
const handleConfPasswordChange =(e)=>{
setConfPassword(e.target.value);
}
// below function will be called when user
// click on submit button .
const handleSubmit=(e)=>{
if(password!=confPassword)
{
// if 'password' and 'confirm password'
// does not match.
alert("password Not Match");
}
else{
// display alert box with user
// 'name' and 'email' details .
alert('A form was submitted with Name :"' + name +
'" ,Age :"'+age +'" and Email :"' + email + '"');
}
e.preventDefault();

}
return (
<div className="App">
<header className="App-header">
```

```
<form onSubmit={(e) => {handleSubmit(e)}}>
{/*when user submit the form , handleSubmit()
function will be called .*/}
<h2> Geeks For Geeks </h2>
<h3> Sign-up Form </h3>
<img src="/gfg.png" />
<label >
Name:
</label><br/>
<input type="text" value={name} required onChange={(e)
=> {handleChange(e)}} /><br/>
{ /*when user write in name input box , handleChange()
function will be called. */}
<label >
Age:
</label><br/>
<input type="text" value={age} required onChange={(e)
=> {handleAgeChange(e)}} /><br/>
{ /*when user write in age input box , handleAgeChange()
function will be called. */}
<label>
Email:
</label><br/>
<input type="email" value={email} required onChange={(e)
=> {handleEmailChange(e)}} /><br/>
{/* when user write in email input box , handleEmailChange()
function will be called.*/}
<label>
Password:
</label><br/>
<input type="password" value={password} required onChange={(e)
=> {handlePasswordChange(e)}} /><br/>
```

```
{/* when user write in password input box ,
handlePasswordChange() function will be called.*/}
<label>
Confirm Password:
</label><br/>
<input type="password" value={confPassword} required onChange={(e)
=> {handleConfPasswordChange(e)}} /><br/>
{/* when user write in confirm password input box ,
handleConfPasswordChange() function will be called.*/}
<input type="submit" value="Submit"/>
</form>
</header>
</div>
);
}

export default App;
```

```
.App {
text-align: center;
}
form {
border:2px solid green;
padding: 30px;
}
img{
height: 120px;
margin-left: 90px;
margin-bottom: 10px;
display: block;
border:1px solid black;
border-radius: 50%;
```

```
}
.App-header {
background-color: white;
min-height: 100vh;
display: flex;
flex-direction: column;
align-items: center;
justify-content: center;
font-size: calc(10px + 2vmin);
color: black;
}
```

10)WHY DO CLASS METHODS NEED TO BOUND TO CLASS INSTANCE, AND HOW YOU CAN AVOID THE NEED FOR BINDING?

This is because **whenever inside a class component when we need to pass a function as props to the child component,** we have to do one of the following: Bind it inside the constructor function. Bind it inline (which can have some performance issues).

To avoid the need for binding we have something introduced in ES6 as arrow functions. **Using the arrow function to call this. setState will lead to avoid the use of bind.**