**Introduction to React and JSX**

**Origins of React**

React is a popular open-source JavaScript library that performs data viewing with the help of HTML. It is also known as ReactJS and React.js, so don’t get confused if you read different notation in different places. As it is developed by in Facebook, it is also popularly known as “Facebook React.js.” Currently, it is maintained by the likes of Instagram, Facebook and community developers that are interested in the library.

So, how does it work? ReactJS works as views using a component based system. The components are specified as custom HTML tags, providing easy usability. React is very useful when it comes to protecting internal components or data flows. The subcomponents cannot be directly affected with external queries, making a good choice for front-end views development. It is also very efficient in updating the HTML document with new data, making it a perfect choice for data-driven web application such as Facebook or Instagram. The library also ensures that a clean separation is made between the different components, ensuring easy maintenance and upgradation in future.

**React.js Syntax**

It is simply a syntax extension of JavaScript. It allows us to directly write HTML in React (within JavaScript code). It is easy to create a template using JSX in React, but it is not a simple template language instead it comes with the full power of JavaScript.

In other words, JSX is an HTML-like syntax used by React that extends ECMAScript so that HTML-like syntax can co-exist with JavaScript/React code. The syntax is used by preprocessors (i.e., transpilers like babel) to transform HTML-like syntax into standard JavaScript objects that a JavaScript engine will parse.

**Overview of JSX**

JSX stands for JavaScript XML. JSX allows us to write HTML in React. JSX makes it easier to write and add HTML in React.

JSX stands for JavaScript syntax extension. It is a JavaScript extension that allows us to describe React's object tree using a syntax that resembles that of an HTML template. It is just an XML-like extension that allows us to write JavaScript that looks like markup and have it returned from a component.

***Differences between JS and JSX***

JS is simply a scripting language, adding functionality into your website. JSX is an addition to the JavaScript syntax which is a mixture of both HTML and JavaScript. Both JS and JSX are interchangeable but JSX makes the code easier to understand for users.

Even though JSX had been around before React, it wouldn't have been nearly as popular without React picking it up. However, we can actually use JSX without React, and it's not that difficult either. The way React works is by configuring your bundler to convert JSX into calls to a create Element function.

HTML is a very important language in web development. Your code is either in HTML originally or compiles to it so browsers can read it. JSX, on the other hand, means JavaScript Syntax Extension or JavaScript XML as some like to put it. It was created as a syntactic sugar for React.

**React Components**

Components are like functions that return HTML elements. Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML. Components come in two types, Class components and Function components.

***Class Component***

A class component must include the extends React.Component statement. This statement creates an inheritance to React.Component, and gives your component access to React.Component's functions. The component also requires a render() method, this method returns HTML.

*Example:*

import React, { Component } from "react";

class ReactClassComponent extends Component {

  render() {

    return (

      <div>

        <h3>React Class Component</h3>

      </div>

    );

  }

}

***Function Component***

Here is the same example as above, but created using a Function component instead.

A Function component also returns HTML, and behaves much the same way as a Class component, but Function components can be written using much less code, are easier to understand, and will be preferred in this tutorial.

*Example:*

import React from 'react'

function ReactFunctionalComponent() {

  return (

      <div>

        <h3>React Functional Component</h3>

      </div>

  )

}

**React State**

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:

import React, { Component } from "react";

class ReactState extends Component {

  constructor(props) {

    super(props);

    this.state = {

      brand: "Maruti",

      model: "Baleno",

      color: "Blue",

      year: 2023,

    };

  }

  render() {

    return (

      <div>

        <h3>React State</h3>

      </div>

    );

  }

}

export default ReactState;

***Using the state Object***

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

import React, { Component } from "react";

class ReactState extends Component {

  constructor(props) {

    super(props);

    this.state = {

      brand: "Maruti",

      model: "Baleno",

      color: "Blue",

      year: 2023,

    };

  }

  render() {

    return (

      <div>

        <h3>React State</h3>

        <h4>Brand: {this.state.brand}</h4>

        <h4>Model: {this.state.model}</h4>

      </div>

    );

  }

}

export default ReactState;

***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).

import React, { Component } from "react";

class ReactState extends Component {

  constructor(props) {

    super(props);

    this.state = {

      brand: "Maruti",

      model: "Baleno",

      color: "Blue",

      year: 2023,

    };

  }

  changeColor = () => {

    this.setState({ color: "Magma Gray" });

  };

  render() {

    return (

      <div>

        <h3>React State</h3>

        <h4>Brand: {this.state.brand}</h4>

        <h4>Model: {this.state.model}</h4>

        <h4>Model: {this.state.color}</h4>

        <button onClick={this.changeColor}>Change Color</button>

      </div>

    );

  }

}

export default ReactState;

**React Props**

Props are arguments passed into React components. Props are passed to components via HTML attributes.

props stands for properties.

React Props are like function arguments in JavaScript and attributes in HTML.

To send props into a component, use the same syntax as HTML attributes

*Example:*

const myElement = <Car brand="Ford" />;

*The component receives the argument as a props object:*

function Car(props) {

return <h2>I am a { props.brand }!</h2>

}

***Props in Functional and Class Components***

|  |  |
| --- | --- |
| function Welcome(props) {  return <h1>Hello, {props.name}</h1>;  } | class Welcome extends React.Component {  render() {  return <h1>Hello, {this.props.name}</h1>;  }  } |

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

const root = ReactDOM.createRoot(document.getElementById('root'));

const element = <Welcome name="Sara" />;

root.render(element);

*Example 1:*

*Garage.js*

import { React } from "react";

const Car = (props) => {

    return (

        <>

            <h1>{props.brand}</h1>

        </>

    )

}

const Garage = () => {

    const carName = "Maruti"

    return (

        <>

            <h3>Car Brand Available at Show Room?</h3>

            <Car brand={carName} />

        </>

    )

}

export { Car, Garage };

*App.js*

import logo from './logo.svg';

import './App.css';

import { Garage } from "./components/Garage";

class App extends React.Component {

  render() {

    return (

      <div className='App'>

        <Garage />

    )

  };

}

export default App;

*Example 2: Props as Object*

*Garage2.js*

import { React } from "react";

const Car2 = (props) => {

    return (

        <>

            <h1>{props.brand.name} {props.brand.model}</h1>

        </>

    )

}

const Garage2 = () => {

    const carInfo = { name: "Toyota", model: "Innova Crysta" }

    return (

        <>

            <h3>Car Brand Available at Show Room?</h3>

            <Car2 brand={carInfo} />

        </>

    )

}

export { Car2, Garage2 };

*App.js*

import logo from './logo.svg';

import './App.css';

import { Garage2 } from "./components/Garage2";

import React from 'react';

class App extends React.Component {

  render() {

    return (

      <div className='App'>

        <Garage2 />

    )

  };

}

export default App;

***Using Props in a Functional Component and in Class Component***

***Props in a Functional Component***

*Prop\_Test.js*

import { React } from "react";

const Propmethod = (props) => {

    return (

        <>

            <h2>Hello {props.firstName} {props.lastName}</h2>

            <button>Click</button>

            {props.children}

        </>

    )

}

export { Propmethod };

*App.js*

import logo from './logo.svg';

import './App.css';

import { Propmethod } from "./components/Prop\_Test";

import React from 'react';

class App extends React.Component {

  render() {

    return (

      <div className='App'>

        <Propmethod firstName="Sharlin" lastName="Lins">

          <p>

            Hai, its my own children content

          </p>

        </Propmethod>

        <Propmethod firstName="John" lastName="Doe">

          <button>Click</button>

        </Propmethod>

      </div>

    )

  };

}

export default App;

***Props in Class Component***

*Prop\_test.js*

import React, { Component } from 'react';

class Propclass extends Component {

    render() {

        return (

            <>

                <p>

                    This is class props example {this.props.firstName} {this.props.lastName}

                </p>

            </>

        );

    }

}

export { Propclass };

*App.js*

import logo from './logo.svg';

import './App.css';

import { Propclass } from "./components/Prop\_Test2";

import React from 'react';

class App extends React.Component {

  render() {

    return (

      <div className='App'>

        <Propclass firstName="Sharlin" lastName="Lins"/>

      </div>

    )

  };

}

export default App;

**React Props Validation**

Properties validation is a useful way to force the correct usage of the components. This will help during development to avoid future bugs and problems, once the app becomes larger. It also makes the code more readable, since we can see how each component should be used.

In this example, we are creating **App** component with all the **props** that we need. **ReactPropsValidation.propTypes** is used for props validation. If some of the props aren't using the correct type that we assigned, we will get a console warning. After we specify validation patterns, we will set **ReactPropsValidation.defaultProps**.

*Example:*

*ReactPropsValidation.js*

import PropTypes from "prop-types";

import React, { Component } from "react";

export class ReactPropsValidation extends Component {

  //   static propTypes = {};

  render() {

    return (

      <div>

        <h3>React Props Validation</h3>

        <table className="table table-sm table-striped">

          <thead>

            <tr>

              <th>

                <p className="float-start">Type</p>

              </th>

              <th>

                <p>Value</p>

              </th>

              <th>

                <p>Valid</p>

              </th>

            </tr>

          </thead>

          <tbody>

            <tr>

              <td>

                <h3 className="float-start">Array</h3>

              </td>

              <td>

                <p>{this.props.propArray}</p>

              </td>

              <td>

                <p>{this.props.propArray ? "true" : "false"}</p>

              </td>

            </tr>

            <tr>

              <td>

                <h3 className="float-start">Boolean</h3>

              </td>

              <td>

                <p>{this.props.propBool}</p>

              </td>

              <td>

                <p>{this.props.propBool ? "true" : "false"}</p>

              </td>

            </tr>

            <tr>

              <td>

                <h3 className="float-start">Function</h3>

              </td>

              <td>

                <p>{this.props.propFunc(2)}</p>

              </td>

              <td>

                <p>{this.props.propFunc(2) ? "true" : "false"}</p>

              </td>

            </tr>

            <tr>

              <td>

                <h3 className="float-start">String</h3>

              </td>

              <td>

                <p>{this.props.propString}</p>

              </td>

              <td>

                <p>{this.props.propString ? "true" : "false"}</p>

              </td>

            </tr>

            <tr>

              <td>

                <h3 className="float-start">Number</h3>

              </td>

              <td>

                <p>{this.props.propNumber}</p>

              </td>

              <td>

                <p>{this.props.propNumber ? "true" : "false"}</p>

              </td>

            </tr>

          </tbody>

        </table>

      </div>

    );

  }

}

ReactPropsValidation.propTypes = {

  propArray: PropTypes.array.isRequired,

  propBool: PropTypes.bool.isRequired,

  propFunc: PropTypes.func,

  propNumber: PropTypes.number,

  propString: PropTypes.string,

};

ReactPropsValidation.defaultProps = {

  propArray: [1, 2, 3, 4, 5],

  propBool: true,

  propFunc: function (x) {

    return x + 5;

  },

  propNumber: 1,

  propString: "PropsValidation",

};

export default ReactPropsValidation;

*App.js*

import "./App.css";

import ReactPropsValidation from "./Components/ReactPropsValidation";

function App() {

  return (

    <div className="App">

      <ReactPropsValidation />

    </div>

  );

}

export default App;

**React Component API**

ReactJS component is a top-level API. It makes the code completely individual and reusable in the application. There are three most important methods available in the React component API.

1. setState()
2. forceUpdate()
3. findDOMNode()

***setState()***

This method is used to update the state of the component. This method does not always replace the state immediately. Instead, it only adds changes to the original state. It is a primary method that is used to update the user interface(UI) in response to event handlers and server responses.

*Syntax:*

this.stateState(object newState[, function callback]);

*Example:*

*ReactCoApiSetState.js*

import React, { Component } from "react";

export class ReactCoApiSetState extends Component {

  constructor(props) {

    super(props);

    this.state = {

      message: "Hello State",

    };

    this.updateSetState = this.updateSetState.bind(this);

  }

  updateSetState = () => {

    this.setState({

      message: "Hello SetState",

    });

  };

  render() {

    return (

      <div>

        <h3>React Composition Api SetState</h3>

        <h4>{this.state.message}</h4>

        <button onClick={this.updateSetState}>SET STATE</button>

      </div>

    );

  }

}

export default ReactCoApiSetState;

*App.js*

import "./App.css";

import ReactCoApiSetState from "./Components/ReactCoApiSetState";

function App() {

  return (

    <div className="App">

      <ReactCoApiSetState />

    </div>

  );

}

export default App;

***forceUpdate()***

This method allows us to update the component manually.

*Syntax:*

Component.forceUpdate(callback);

*Example:*

*ReactCoApiForceUpdate.js*

import React, { Component } from "react";

export class ReactCoApiForceUpdate extends Component {

  constructor(props) {

    super(props);

    this.forceUpdateState = this.forceUpdateState.bind(this);

  }

  forceUpdateState = () => {

    this.forceUpdate();

  };

  render() {

    return (

      <div>

        <h3>React Composition Api ForceUpdate</h3>

        <h4>Random Number: {Math.random()}</h4>

        <button onClick={this.forceUpdateState}>ForceUpdate</button>

      </div>

    );

  }

}

export default ReactCoApiForceUpdate;

*App.js*

import "./App.css";

import ReactCoApiForceUpdate from "./Components/ReactCoApiForceUpdate";

function App() {

  return (

    <div className="App">

      <ReactCoApiForceUpdate />

    </div>

  );

}

export default App;

***findDOMNode()***

For DOM manipulation, you need to use ReactDOM.findDOMNode() method. This method allows us to find or access the underlying DOM node.

*Syntax:*

ReactDOM.findDOMNode(component);

*Example:*

*ReactCoApiFindDom.js*

import React, { Component } from "react";

import ReactDOM from "react-dom";

class ReactCoApiFindDom extends Component {

  doFind() {

    var node = ReactDOM.findDOMNode(this);

    node.style.border = "3px solid red";

  }

  render() {

    return (

      <div>

        <h3>React Composition Api findDOMNode</h3>

        <ul>

          <li>Apple</li>

          <li>Apricot</li>

          <li>Banana</li>

          <li>

            <button onClick={() => this.doFind()}>Find Root Node</button>

          </li>

        </ul>

      </div>

    );

  }

}

export default ReactCoApiFindDom;

*App.js*

import "./App.css";

import ReactCoApiFindDom from "./Components/ReactCoApiFindDom";

function App() {

  return (

    <div className="App">

      <ReactCoApiFindDom />

    </div>

  );

}

export default App;

**React Component Life Cycle**

In ReactJS, every component creation process involves various lifecycle methods. These lifecycle methods are termed as component's lifecycle. There are four phases available they are

1. Initial Phase
2. Mounting Phase
3. Updating Phase
4. Unmounting Phase

Each phase consists of some lifecycle methods that are specific to the particular phase.

**1. Initial Phase**

It is the birth phase of the lifecycle of a ReactJS component. Here, the component starts its journey on a way to the DOM. In this phase, a component contains the default Props and initial State. These default properties are done in the constructor of a component. The initial phase only occurs once and consists of the following methods.

* getDefaultProps()
* getInitialState()

***getDefaultProps()***

This is used to specify the default value of *this.props*. It is invoked before the creation of the component or any props from the parent is passed into it.

***getInitialState()***

This is used to specify the default value of *this.props*. It is invoked before the creation of the component.

**2. Mounting Phase**

In the mounting phase, the instance of a component is created and inserted into the DOM. It consists of the following methods.

* componentWillMount()
* componentDidMount()
* render()

***componentWillMount()***

This method is invoked before the component get rendered in the DOM. When we call the *setState()* inside this method the component will not re-render.

***componentDidMount()***

This method is invoked after the component get rendered and placed in the DOM.

***render()***

This method is defined in every component. It is used to return the *HTML* element, if no need to render anything we can return *null* or *false* value.

**3. Updating Phase**

In the Updating phase, we get the *new* *props* and change *state.* This phase also allows to handle user interaction and provide communication with the components hierarchy. The main aim of this phase is to ensure that the component is displaying the latest version of itself. This phase consists of the following methods.

* componentWillRecieveProps()
* shouldComponentUpdate()
* componentWillUpdate()
* render()
* componentDidUpdate()

***componentWillRecieveProps()***

This method is invoked when a component receives *new props*. If you want to update the state in response to *prop changes*, you should compare *this.props* and *newProps* to perform state transition by using ***this.setState()*** method.

***shouldComponentUpdate()***

This method is invoked when a component decides any changes or updation to the DOM. It allows you to control the component's behaviour of updating itself. If this method returns true, the component will update otherwise, the component will skip the updating.

***componentWillUpdate()***

This method is invoked just before the component updating occurs. Here, you can't change the component state by invoking ***this.setState()*** method. This method will not be called, if ***shouldComponentUpdate()*** returns false.

***render()***

This method is invoked to examine *this.props* and *this.state* and return one of the following types: React elements, Arrays and fragments, Booleans or null, String and Number. If ***shouldComponentUpdate()*** returns false, the code inside ***render()*** will be invoked again to ensure that the component displays itself properly.

***componentDidUpdate()***

This method is invoked immediately after the component updating occurs. In this method, you can write any code inside which you want to execute once the updating occurs. This method is not invoked for the initial render.

**4. Unmounting Phase**

The Unmounting phase is the final phase of the react component lifecycle. It is called when a component instance is destroyed and unmounted from the DOM. This phase consists of the only one method.

* componentWillUnmount()

***componentWillUnmount()***

This method is invoked immediately before a component is destroyed and unmounted permanently. It performs any necessary *cleanup* related task such as invalidating timers, event listener, cancelling network requests, or cleaning up DOM elements. If a component instance is unmounted, you cannot mount it again.

*Example:*

*ReactLifeCycle.js*

import React, { Component } from "react";

class ReactLifeCycle extends Component {

  constructor(props) {

    super(props);

    this.state = {

      hello: "REACTJS",

    };

    this.changeState = this.changeState.bind(this);

  }

  render() {

    return (

      <div>

        <h3>ReactJs Component LifeCycle</h3>

        <h4>{this.state.hello}</h4>

        <button onClick={this.changeState}>Click Here</button>

      </div>

    );

  }

  componentWillMount() {

    console.log("Component Will Mount");

  }

  componentDidMount() {

    console.log("Component Did Mount");

  }

  changeState() {

    this.setState({ hello: "React Life Cycle" });

  }

  componentWillReceiveProps(newProps) {

    console.log("Component Will Receive Props");

  }

  shouldComponentUpdate(newProps, newState) {

    return true;

  }

  componentWillUpdate(newProps, newState) {

    console.log("Component Will Update");

  }

  componentDidUpdate(prevProps, prevState) {

    console.log("Component Did Update");

  }

  componentWillUnmount() {

    console.log("Component Will Unmount");

  }

}

export default ReactLifeCycle;

*App.js*

import "./App.css";

import ReactLifeCycle from "./Components/ReactLifeCycle";

function App() {

  return (

    <div className="App">

      <ReactLifeCycle />

    </div>

  );

}

export default App;

**React Lists**

In React, we render lists with some type of loop. The JavaScript map() array method is generally the preferred method.

*Example: Map()*

*ReactLists.js*

import React from 'react'

const Car = (props) => {

    return <li>{props.brand}</li>

}

function ReactLists() {

    const cars = ['Maruti', 'Tata', 'Ford']

    return (

        <div>

            <ul>

                {cars.map((car) => <Car brand={car} />)}

            </ul>

        </div>

    )

}

export default ReactLists

*App.js*

import './App.css';

import ReactLists from './Components/ReactLists';

function App() {

  return (

    <div className="App">

      <ReactLists/>

    </div>

  );

}

export default App;

**Keys**

Keys allow React to keep track of elements. This way, if an item is updated or removed, only that item will be re-rendered instead of the entire list. Keys need to be unique to each sibling. But they can be duplicated globally.

*Example: Keys()*

*ReactLists.js*

import React from 'react'

const Car = (props) => {

    return <li>{props.brand}</li>

}

function ReactLists() {

    const cars = [

        { id: 1, brand: 'Maruti' },

        { id: 2, brand: 'Tata' },

        { id: 3, brand: 'Ford' }

    ]

    return (

        <div>

            <ul>

                {cars.map((car) => <Car key={car.id} brand={car.brand} />)}

            </ul>

        </div>

    )

}

export default ReactLists

*App.js*

import './App.css';

import ReactLists from './Components/ReactLists';

function App() {

  return (

    <div className="App">

      <ReactLists/>

    </div>

  );

}

export default App;

**React Forms**

Just like in HTML, React uses forms to allow users to interact with the web page.

Handling forms is about how you handle the data when it changes value or gets submitted.

In HTML, form data is usually handled by the DOM. In React, form data is usually handled by the components.

When the data is handled by the components, all the data is stored in the component state. You can control changes by adding event handlers in the onChange attribute. We can use the useState Hook to keep track of each inputs value and provide a "single source of truth" for the entire application.

You can control the submit action by adding an event handler in the onSubmit attribute for the <form>:

*Example:*

*ReactLoginForm*.*js*

import React from 'react'

import { useState } from 'react';

const ReactLoginForm = () => {

    const [name, setName] = useState('')

    const [age, setAge] = useState('')

    const [email, setEmail] = useState('')

    const [password, setPassword] = useState('')

    const [confPassword, setConfPassword] = useState('')

    const handleName = (event) => {

        setName(event.target.value)

    }

    const handleAge = (event) => {

        setAge(event.target.value)

    }

    const handleEmail = (event) => {

        setEmail(event.target.value)

    }

    const handlePassword = (event) => {

        setPassword(event.target.value)

    }

    const handleConfPassword = (event) => {

        setConfPassword(event.target.value)

    }

    const HandleSubmit = (event) => {

        if (password !== confPassword) {

            alert("Passwords Didn't Match!")

        }

        alert(`The Form Submitted with Name : ${name}, Age: ${age}, Email: ${email}`)

        event.preventDefault();

    }

    return (

        <div>

            <form onSubmit={HandleSubmit}>

                <table>

                    <tr>

                        <td className='text-start'>Name: </td>

                        <td><input className='form-control form-control-sm mt-0' type="text" value={name} onChange={handleName} /></td>

                    </tr>

                    <tr>

                        <td className='text-start'>Age: </td>

                        <td><input className='form-control form-control-sm mt-0' type="text" value={age} onChange={handleAge} /></td>

                    </tr>

                    <tr>

                        <td className='text-start'>Email: </td>

                        <td><input className='form-control form-control-sm mt-0' type="email" value={email} onChange={handleEmail} /></td>

                    </tr>

                    <tr>

                        <td className='text-start'>Password: </td>

                        <td><input className='form-control form-control-sm mt-0' type="password" value={password} onChange={handlePassword} /></td>

                    </tr>

                    <tr>

                        <td className='text-start'>Confirm Password: </td>

                        <td><input className='form-control form-control-sm mt-0' type="password" value={confPassword} onChange={handleConfPassword} /></td>

                    </tr>

                    <tr>

                        <td colSpan="1"></td>

                        <td><button className="btn btn-outline-primary" type='submit'>Submit</button></td>

                    </tr>

                </table>

            </form>

        </div>

    )

}

export default ReactLoginForm

*App.js*

import './App.css';

import ReactLoginForm from './Components/ReactLoginForm';

function App() {

  return (

    <div className="App">

      <ReactLoginForm />

    </div>

  );

}

export default App;

**React Event Handling**

Just like HTML DOM events, React can perform actions based on user events. React has the same events as HTML: click, change, mouseover etc.

***Adding Events***

React events are written in camelCase syntax:

onClick instead of onclick.

React event handlers are written inside curly braces:

onClick={shoot}  instead of onClick="shoot()".

|  |  |
| --- | --- |
| **React** | **Html** |
| *<button onClick={shoot}>Take the Shot!</button>* | *<button onclick="shoot()">Take the Shot!</button>* |

*Example:*

ButtonClick.js

import React from 'react'

function ButtonClick() {

    const msg = () => {

        alert("Hai, Clicked the Button!!")

    }

    return (

        <button onClick={msg}>Click Me</button>

    )

}

export default ButtonClick

App.js

import React from 'react';

import ButtonClick from './components/ButtonClick';

class App extends React.Component {

  render() {

    return (

      <div className='App'>

        <ButtonClick/>

      </div>

    )

  };

}

export default App;

*Example 2: Showing different time zone on button click*

*TimeZone.js*

import React from 'react'

function TimeZone() {

    const india = () => {

        var time = new Date().toLocaleString('en-GB', { timeZone: 'Asia/Kolkata', dateStyle: 'full', timeStyle: 'full' })

        alert(time)

    }

    const london = () => {

        var time = new Date().toLocaleString('en-GB', { timeZone: 'Europe/London', dateStyle: 'full', timeStyle: 'full' })

        alert(time)

    }

    const america = () => {

        var time = new Date().toLocaleString('en-GB', { timeZone: 'America/New\_York', dateStyle: 'full', timeStyle: 'full' })

        alert(time)

    }

    const australia = () => {

        var time = new Date().toLocaleString('en-GB', { timeZone: 'Australia/Sydney', dateStyle: 'full', timeStyle: 'full' })

        alert(time)

    }

    return (

        <>

            <button onClick={india}>Click for Indian Time</button>

            <button onClick={london}>Click for London Time</button>

            <button onClick={america}>Click for America Time</button>

            <button onClick={australia}>Click for Australia Time</button>

        </>

    )

}

export default TimeZone

*App.js*

import logo from './logo.svg';

import './App.css';

import React from 'react';

import TimeZone from './components/TimeZone';

class App extends React.Component {

  render() {

    return (

      <div className='App'>

        <TimeZone />

      </div>

    )

  };

}

export default App;

*Example 3: Showing different time zone using Option button*

*TimeZone2.js*

import React from 'react'

function TimeZone2() {

    const india = () => {

        var time = new Date().toLocaleString('en-GB', { timeZone: 'Asia/Kolkata', dateStyle: 'full', timeStyle: 'full' })

        alert(time)

    }

    const london = () => {

        var time = new Date().toLocaleString('en-GB', { timeZone: 'Europe/London', dateStyle: 'full', timeStyle: 'full' })

        alert(time)

    }

    const america = () => {

        var time = new Date().toLocaleString('en-GB', { timeZone: 'America/New\_York', dateStyle: 'full', timeStyle: 'full' })

        alert(time)

    }

    const australia = () => {

        var time = new Date().toLocaleString('en-GB', { timeZone: 'Australia/Sydney', dateStyle: 'full', timeStyle: 'full' })

        alert(time)

    }

    return (

        <>

            <input type="radio" name="option" onClick={india} />Click for Indian Time<br />

            <input type="radio" name="option" onClick={london} />Click for London Time<br />

            <input type="radio" name="option" onClick={america} />Click for America Time<br />

            <input type="radio" name="option" onClick={australia} />Click for Australia Time

        </>

    )

}

export default TimeZone2

*App.js*

import logo from './logo.svg';

import './App.css';

import React from 'react';

import TimeZone2 from './components/TimeZone2';

class App extends React.Component {

  render() {

    return (

      <div className='App'>

        <TimeZone2 />

      </div>

    )

  };

}

export default App;

**React Ref**

Refs are the shorthand, similar to the React keys. They are used for references in React, to store a reference to particular DOM nodes or React elements, to access React DOM nodes or React elements, to interact with React DOM nodes or React elements and to change the value of a child component, without using props.

Thus it can be used when we need DOM measurements or if there is a need to trigger imperative animations. It can also be used while integrating with third-party DOM libraries and in callbacks. However, its use should be avoided for declarative actions and for overuse purposes.

***Creating Refs***

*React.createRef()* method is used to create React Ref.

***Accessing Refs***

const node = this.callRef.current;

*Example:*

*ReactRef.js*

import React, { Component } from "react";

export class ReactRef extends Component {

  constructor(props) {

    super(props);

    this.callRef = React.callRef();

    this.refInput = this.refInput.bind(this);

  }

  refInput() {

    this.callRef.current.focus();

  }

  render() {

    return (

      <div>

        <h3>React Ref</h3>

        <h1>Hello World</h1>

        <input type="text" ref={this.callRef} />

        <input type="button" value="Enter text" onClick={this.refInput} />

      </div>

    );

  }

}

export default ReactRef;

*App.js*

import "./App.css";

import ReactRef from "./Components/ReactRef";

function App() {

  return (

    <div className="App">

      <ReactRef />

    </div>

  );

}

export default App;