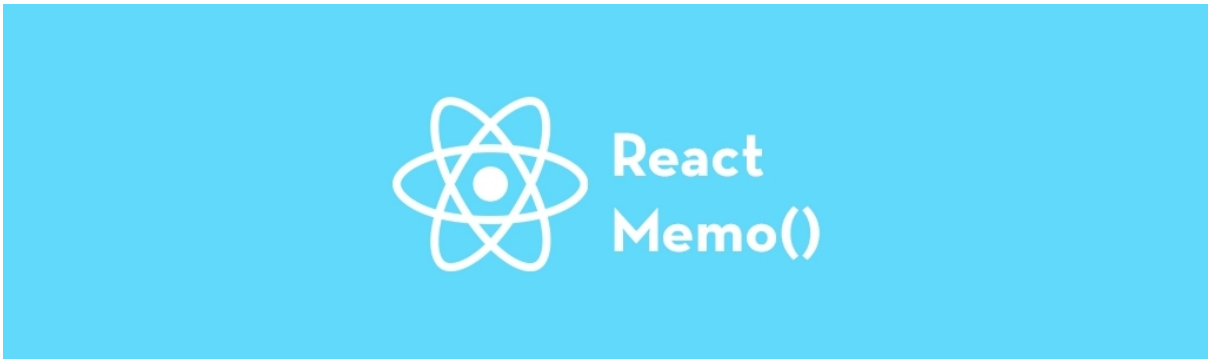
Memo



What is React Memo?Anchor

Components in React are designed to re-render whenever the state or props value changes. However, this can impact your application's performance because, even if the change is only intended to affect the parent component, every other child component attached to the parent component will re-render. When a parent component re-renders, so do all of its child components.

React Memo is a **higher-order component** that wraps around a component to memoize the rendered output and avoid unnecessary renderings. This improves performance because it memoizes the result and skips rendering to reuse the last rendered result.

There are two ways you can wrap your component with React.memo(). It is either you wrap the actual component directly without having to create a new variable to store the memoized component:

```
const myComponent = React.memo((props) => {
   /* render using props */
});

export default myComponent;
```

Another option is to create a new variable to store the memoized component and then export the new variable:

```
const myComponent = (props) => {
   /* render using props */
};

export const MemoizedComponent = React.memo(myComponent);
```

In the example above, myComponent outputs the same content as MemoizedComponent, but the difference between both is that MemoizedComponent render is memoized. This means that this component will only re-render when the props change.

Note: A memoized component will only re-render when there is a change in props value or when the state and context of the component change.

When to use React MemoAnchor

You now know what it means to memoize a component and the advantages of optimization. But this doesn't mean you should memoize all your components to ensure maximum performance optimization .

It is important to know when and where to memoize your component else it would not fulfill its purpose. For example, React Memo is used to avoid unnecessary re-renders when there is no change to the state or context of your component. If the state and content of your component will ALWAYS change, React Memo becomes useless. Here are other points:

- Use React Memo if your component will render quite often.
- Use it when your component often renders with the same props. This happens to child components who are forced to re-render with the same props whenever the parent component renders.
- Use it in pure functional components alone. If you are using a class component, use the React.PureComponent.
- Use it if your component is big enough (contains a decent amount of UI elements) to have props equality check.

How to use React MemoAnchor

At this point, you now understand when to use React Memo and what it does. The major point is that you should use it when your component often renders with the same props.

Suppose you have a React application such as a Todo application in which you decide to break up the application into separate components to ensure it is easy to understand and use. You will have the parent component that holds your todo array in a state. The todo array will be passed as a prop to the Todo component:

```
// App.js
import React, { useState } from 'react';
import Todo from './Todo';

const App = () => {
   console.log('App component rendered');
   const [todo, setTodo] = useState([
      { id: 1, title: 'Read Book' },
      { id: 2, title: 'Fix Bug' },
   ]);
   const [text, setText] = useState('');

   const addTodo = () => {
      let newTodo = { id: 3, title: text };
      setTodo([...todo, newTodo]);
```

```
  };

  return (
    <div>
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
      />
      <button type="button" onClick={addTodo}>
        Add todo
      </button>
      <Todo list={todo} />
    </div>
  );
};

export default App;
```

In the component above, there is a state to hold all the todo in an array, and then there is a form you would use to add a new todo to the array. Notice that at the beginning of the component, a console.log() statement will be implemented once your application renders.

The todo items are to be passed as props to the Todo component, which has been imported. This means that the Todo component is a child component of the App component. In the Todo component, the todo array that has been passed as a prop named list will be iterated so that you can access each item in the array and display:

```
// Todo.js
import React from 'react';
import TodoItem from './TodoItem';

const Todo = ({ list }) => {
  console.log('Todo component rendered');
  return (
    <ul>
      {list.map((item) => (
        <TodoItem key={item.id} item={item} />
      ))}
    </ul>
  );
};
export default Todo;
```

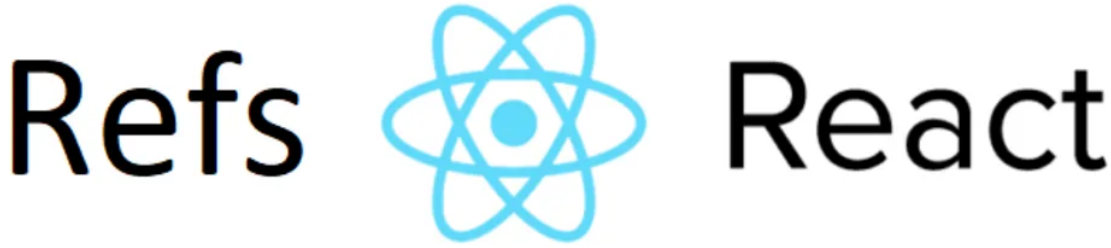
In the code above, a console.log() statement will display a text to show when the Todo component renders or re-renders. In the component above, each todo item is passed as a prop to a TodoItem component. This is done to properly explain what happens and when you need to memoize a component:

```
//TodoItem.js
const TodoItem = ({ item }) => {
   console.log('TodoItem component rendered');
   return <li>{item.title}</li>;
};
export default TodoItem;
```

The component above also has a console.log statement to help you know when it renders and re-renders. When you run your application and check the console, you will notice that all three components render.

Our component renders four times, once for the parent component (App.js), once for the Todo component, and finally, the TodoItem component renders twice because it has two elements. This means altogether, the application renders four times. This is normal for the initial render. But when you change something in the parent component that doesn't affect the child components, only the parent component should be rendered.

Introduction to Refs: Refs,



Refs is the shorthand used for references in React. It is similar to keys in React. It is an attribute which makes it possible to store a reference to particular DOM nodes or React elements. It provides a way to access React DOM nodes or React elements and how to interact with it. It is used when we want to change the value of a child component, without making the use of props.

When to Use Refs

Refs can be used in the following cases:

- When we need DOM measurements such as managing focus, text selection, or media playback.
- It is used in triggering imperative animations.
- When integrating with third-party DOM libraries.
- It can also be used in callbacks.

When to not use Refs

- Its use should be avoided for anything that can be done declaratively. For example, instead of using open() and close() methods on a Dialog component, you need to pass an isOpen prop to it.
- You should have to avoid overuse of the Refs.

How to create Refs

In React, Refs can be created by using React.createRef(). It can be assigned to React elements via the ref attribute. It is commonly assigned to an instance property when a component is created, and then can be referenced throughout the component.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.callRef = React.createRef();
  }
  render() {
    return <div ref={this.callRef} />;
  }
```

```
}
```

How to access Refs

In React, when a ref is passed to an element inside the render method, a reference to the node can be accessed via the current attribute of the ref.

```
const node = this.callRef.current;
```

Refs current Properties

The ref value differs depending on the type of the node:

When the ref attribute is used in an HTML element, the ref created with React.createRef() receives the underlying DOM element as its current property.

If the ref attribute is used on a custom class component, then the ref object receives the mounted instance of the component as its current property.

The ref attribute cannot be used on function components because they don't have instances.

Add Ref to DOM elements

In the below example, we are adding a ref to store the reference to a DOM node or element.

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class App extends React.Component {
   constructor(props) {
      super(props);
      this.callRef = React.createRef();
      this.addingRefInput = this.addingRefInput.bind(this);
   }

   addingRefInput() {
      this.callRef.current.focus();
   }

   render() {
      return (
         <div>
            <h2>Adding Ref to DOM element</h2>
            <input
               type="text"
               ref={this.callRef} />
            <input
               type="button"
               value="Add text input"
```

```
            onClick={this.addingRefInput}
          />
        </div>
      );
   }
}
export default App;
```

Add Ref to Class components

In the below example, we are adding a ref to store the reference to a class component.

Example

```
import React, { Component } from 'react';
import { render } from 'react-dom';

function CustomInput(props) {
   let callRefInput = React.createRef();

   function handleClick() {
      callRefInput.current.focus();
   }

   return (
      <div>
         <h2>Adding Ref to Class Component</h2>
         <input
            type="text"
            ref={callRefInput} />
         <input
            type="button"
            value="Focus input"
            onClick={handleClick}
         />
      </div>
   );
}
class App extends React.Component {
   constructor(props) {
      super(props);
      this.callRefInput = React.createRef();
   }

   focusRefInput() {
      this.callRefInput.current.focus();
   }
```

```
  render() {
    return (
      <CustomInput ref={this.callRefInput} />
    );
  }
}
export default App;
```

Callback refs

In react, there is another way to use refs that is called "callback refs" and it gives more control when the refs are set and unset. Instead of creating refs by createRef() method, React allows a way to create refs by passing a callback function to the ref attribute of a component. It looks like the code below.

```
<input type="text" ref={element => this.callRefInput = element} />
```

The callback function is used to store a reference to the DOM node in an instance property and can be accessed elsewhere. It can be accessed as below:

```
this.callRefInput.value
```

The example below helps to understand the working of callback refs.

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.callRefInput = null;

    this.setInputRef = element => {
      this.callRefInput = element;
    };

    this.focusRefInput = () => {
      //Focus the input using the raw DOM API
      if (this.callRefInput) this.callRefInput.focus();
    };
  }

  componentDidMount() {
```

```
    //autofocus of the input on mount
    this.focusRefInput();
  }

  render() {
    return (
      <div>
        <h2>Callback Refs Example</h2>
        <input
          type="text"
          ref={this.setInputRef}
        />
        <input
          type="button"
          value="Focus input text"
          onClick={this.focusRefInput}
        />
      </div>
    );
  }
}
export default App;
```

In the above example, React will call the "ref" callback to store the reference to the input DOM element when the component mounts, and when the component unmounts, call it with null. Refs are always up-to-date before the componentDidMount or componentDidUpdate fires. The callback refs passed between components is the same as you can work with object refs, which is created with React.createRef().

Forwarding Refs:

The forwardRef method in React allows parent components to move down (or “forward”) refs to their children. ForwardRef gives a child component a reference to a DOM entity created by its parent component in React. This helps the child to read and modify the element from any location where it is used.

How does forwardRef work in React?

In React, parent components typically use props to transfer data down to their children. Consider you make a child component with a new set of props to change its behavior. We need a way to change the behavior of a child component without having to look for the state or re-rendering the component. We can do this by using refs. We can access a DOM node that is represented by an element using refs. As a result, we will make changes to it without affecting its state or having to re-render it.

When a child component needs to refer to its parent’s current node, the parent component must have a way for the child to receive its ref. The technique is known as ref forwarding.

Syntax:

```
React.forwardRef((props, ref) => {})
```

Parameters: It takes a function with props and ref arguments.
Return Value: This function returns a JSX Element.

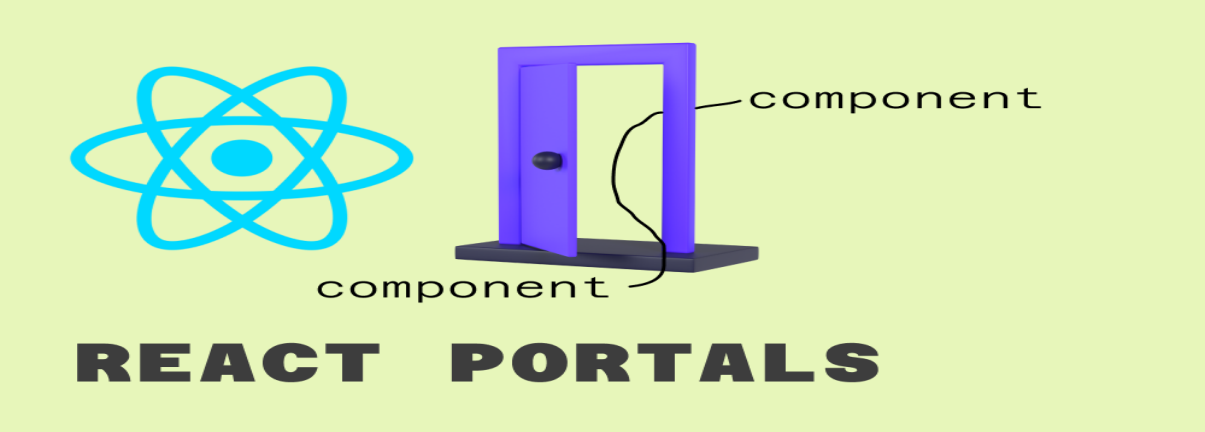
```
import React from 'react'

class App extends React.Component {
  constructor(props) {
    super(props)
    this.aRef = React.createRef()
  }
  render() {
    return (
      <>
        <Counter ref={this.aRef} />
        <button onClick={() => { console.log(this.aRef) }}>
          Ref
        </button>
      </>
    )
  }
}

const Counter = React.forwardRef((props, ref) => {
  class Counter extends React.Component {
    constructor(props) {
      super(props)
      this.state = {
        count: 0
      }
    }
    render() {
      return (
        <div>
          Count: {this.state.count}
          <button ref={ref} onClick={() => this.setState(
            { count: this.state.count + 1 })}>
            Increment
          </button>
        </div>
      )
    }
  }
```

```
    return <Counter />
})

export default App
```

Explanation: Since the counter is within the function, it can access the props and ref parameters using a closure. The counter is rendered and returned. The ref passed to the Counter component is set to the value of the button element's ref attribute. The counter element's ref attribute will now be set to refer to the button element.

The Counter component is rendered by the App component. It starts by creating a ref this.aRef is assigned to the Counter component's ref attribute as a value. We've included a button that logs the value of this.aRef. The this.aRef will hold the HTMLButtonElement of the Incr button in the Counter component. Clicking on the Ref button will confirm that It will log this.aRef which will log the object of the Incr button HTMLButtonElement. It didn't log the instance of the Counter because the Counter component forwarded it to its child component, the Increment button.

React portals:



React portals come up with a way to render children into a DOM node that occurs outside the DOM hierarchy of the parent component. The portals were introduced in the React 16.0 version.

So far we were having one DOM element in the HTML into which we were mounting our react application, i.e., the root element of our index.html in the public folder. Basically, we mount our App component onto our root element. It is almost a convention to have a div element with the id of root to be used as the root DOM element. If you take a look at the browser in the DOM tree every single React component in our application falls under the root element, i.e., inside this statement.

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

But React Portals provide us the ability to break out of this dom tree and render a component onto a dom node that is not under this root element. Doing so breaks the convention where a component needs to be rendered as a new element and follow a parent-child hierarchy. They are commonly used in modal dialog boxes, hovercards, loaders, and popup messages.

Syntax:

```
ReactDOM.createPortal(child, container)
```

Parameters: Here the first parameter is a child which can be a React element, string, or a fragment and the second parameter is a container which is the DOM node (or location) lying outside the DOM hierarchy of the parent component at which our portal is to be inserted.

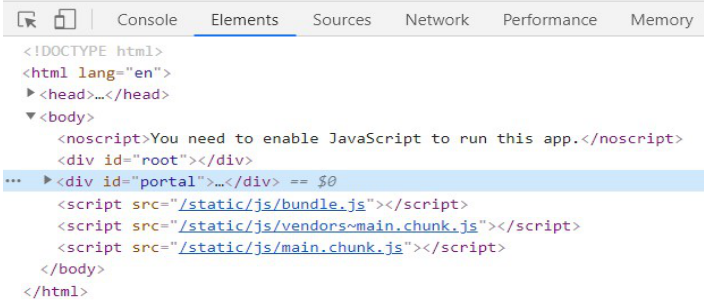
Importing: To create and use portals you need to import ReactDOM as shown below.

Example

```
//App.js
import ReactDOM from 'react-dom'
function App() {
  // Creating a portal
  return ReactDOM.createPortal(
```

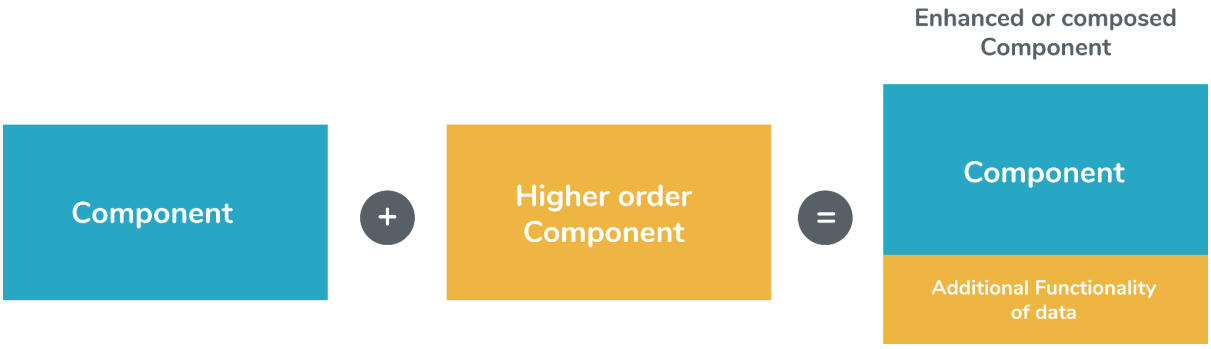
```
   <h1>Portal demo</h1>,
   document.getElementById('portal')
 )
}
export default App;
```

```
<!-- index.htm–>
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="utf-8" />
  <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <meta name="theme-color" content="#000000" />
  <meta
   name="description"
   content="Web site created using create-react-app"
  />
  <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
  <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
  <title>React App</title>
 </head>
 <body>
  <noscript>
   You need to enable JavaScript to run this app.
  </noscript>
  <div id="root"></div>
      <!-- new div added to access the child component -->
  <div id="portal"></div>
 </body>
</html>
```



Explanation: Here, we can see that our <h1> tag Portal Demo is under the newly created portal DOM node, not the traditional root DOM node. It clearly tells us how React Portal provides the ability to break out of the root DOM tree and renders a component/element outside the parent DOM element.

Components: Higher Order Components



It is also known as HOC. In React, HOC is an advanced technique for reusing component logic. It is a function that takes a component and returns a new component. According to the official website, it is not the feature(part) in React API, but a pattern that emerges from React compositional nature. They are similar to JavaScript functions used for adding additional functionalities to the existing component.

A higher order component function accepts another function as an argument. The map function is the best example to understand this. The main goal of this is to decompose the component logic into simpler and smaller functions that can be reused as you need.

Syntax:

```
const NewComponent = higherOrderComponent(WrappedComponent);
```

We know that a component transforms props into UI, and a higher-order component converts a component to another component and allows to add additional data or functionality into this. Hocs are common in third-party libraries. The examples of HOCs are Redux's connect and Relay's createFragmentContainer.

Now, we can understand the working of HOCs from the below example.

```
//Function Creation
function add(a, b) {
   return a + b
}
function higherOrder(a, addReference) {
   return addReference(a, 20)
}
//Function call
higherOrder(30, add) // 50
```

In the above example, we have created two functions add() and higherOrder(). Now, we provide the add() function as an argument to the higherOrder() function. For invoking, rename it addReference in the higherOrder() function, and then invoke it.

Here, the function you are passing is called a callback function, and the function where you are passing the callback function is called a higher-order(HOCs) function.

Example

Create a new file with the name HOC.js. In this file, we have made one function HOC. It accepts one argument as a component. Here, that component is App.

```
//HOC.js
import React, { Component } from 'react';
export default function Hoc(HocComponent) {
   return class extends Component {
      render() {
         return (
            <div>
               <HocComponent></HocComponent>
            </div>

         );
      }
   }
}
```

Now, include the HOC.js file into the App.js file. In this file, we need to call the HOC function.

```
App = Hoc(App);
```

The App component is wrapped inside another React component so that we can modify it. Thus, it becomes the primary application of the Higher-Order Components.

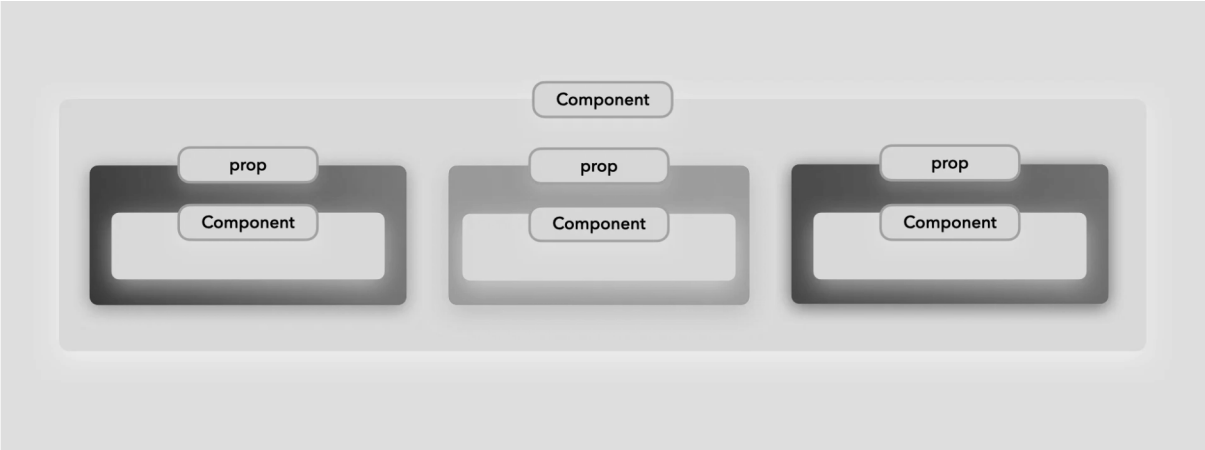
```
//App.js
import React, { Component } from 'react';
import Hoc from './HOC';

class App extends Component {
  render() {
    return (
      <div>
        <h2>HOC Example</h2>
        Geetanjali provides best CS tutorials.
      </div>
    )
  }
}
App = Hoc(App);
export default App;
```

Higher-Order Component Conventions
Do not use HOCs inside the render method of a component.

The static methods must be copied over to have access to them. You can do this using the hoist-non-react-statics package to automatically copy all non-React static methods.

HOCs do not work for refs as 'Refs' does not pass through as a parameter or argument. If you add a ref to an element in the HOC component, the ref refers to an instance of the outermost container component, not the wrapped component.

Props Again!: Rendering Props and Context



Render Props in React is something where a component's prop is assigned a function and that is called in the render method of the component. Calling the function will return a React element or component. Third-party libraries like React Router and Downshift use this approach.

A render prop is a function prop that the component uses to know what to render. Through this, we can also pass a state to another.

Example

Create a Movement.js file and code as below,

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

export class Movement extends Component {

  constructor(props) {
    super(props)

    this.state = {
      x: 0,
      y: 0
    }
  }

  onMouseMove = (event) => {
    this.setState({
      x: event.clientX,
      y: event.clientY
    })
  }

  render() {
    const { x, y } = this.state
    return (
```

```
        <div style={{ height: '100%' }} onMouseMove={this.onMouseMove}>
          <h1> The current mouse position is ({x},{y})</h1>
        </div>
      )
    }
}

export default Movement
```

Now add this Movement.js in App.js,

```
import React from 'react';
import './App.css';
import './css/custom.css';
import Movement from './components/Movement'

function App() {
  return (
    <div>
      <Movement></Movement>
    </div>
  );
}

export default App;
```

Now let's go further with the demo, we will create a component named Monkey in which on moving the mouse, the monkey will also move following the banana.

Change the code of Movement.js as below.

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'

export class Movement extends Component {

  static propTypes = {
    render: PropTypes.func.isRequired
  }

  state = { x: 0, y: 0 }

  onMouseMove = (event) => {
    this.setState({
      x: event.clientX,
      y: event.clientY
```

```
      })
   }

   render() {
      return (
         <div onMouseMove={this.onMouseMove}>
            {this.props.render(this.state)}
         </div>
      )
   }
}
export default Movement
```

Now, create Monkey.js with the below code. Below, we also need 2 images of monkey and banana respectively which are uploaded in the source code of Demo2.

```
import React, { Component } from 'react'
import monkey from '../images/monkey.jpg'
import banana from '../images/banana.jpg'

export class Monkey extends Component {
   render() {
      const { x, y } = this.props.Movement

      return (
         <div>
            <img src={monkey} alt='monkey' style={{ position: 'relative', left: x, right: y,
height: 150 }}></img>
            <img src={banana} alt='banana' style={{ position: 'relative', left: (x + 10), right: (y
+ 10), height: 30 }}></img>
         </div>
      )
   }
}

export default Monkey
```

Now, in the final step, call Monkey.js in App.js.

```
import React from 'react';
import './App.css';
import './css/custom.css';
import Movement from './components/Movement'
import Monkey from './components/Monkey'

function App() {
```

```
  return (
    <div>
      <Movement render={({ x, y }) => (
        <Monkey Movement={{ x, y }} />
      )} />

    </div>
  );
}

export default App;
```

In the above program, on moving the mouse, the monkey will also move. The above demo is quite interesting and lets us allow other animation to be created using react-motion API.

Context

Context in React.js is the concept of passing data through a component tree without passing props down manually to each level.

In basic React applications, to pass data from parent to child is done using props but this is a heavy approach if data needs to be passed on multiple levels like passing username or theme required by most of the components in the application. Context lets you provide the functionality to share values among components without explicitly passing props through every level of the component tree.

Mainly context is used when the current authentication, themes, or preferred language need to be passed to the child levels when not required to pass props.

To get data using context we need to follow 3 steps,

- We need to first create the context.
- Then we need to provide value to the context.
- Lastly, we need to consume the value of context.

Now let's have a look at the demo,

For Example - In the current demo, we will see how to use username in child components using context.

First, we will create a new file namely UserContext.js. This is the first step where we are

```
import React from 'react'

const UserContext = React.createContext()

const UserProvider = UserContext.Provider
const UserConsumer = UserContext.Consumer

export {UserProvider,UserConsumer}
```

Now, we will create a nested structure in which I will create 3 level components Master Component -> Home Component -> Profile Component.

App.js in which we will execute second step we need to provide value to consumer,

```
import React from 'react';
import './App.css';
import './css/custom.css';
import MasterComponent from './components/MasterComponent'
import { UserProvider } from './components/UserContext';

function App() {
  return (
```

```
      <div className="App">
        <UserProvider value="New User">
          <MasterComponent />
        </UserProvider>
      </div>
    );
}

export default App;
```

MasterComponent.js

```
import React, { Component } from 'react'
import HomeComponent from './HomeComponent'
class MasterComponent extends Component {
   render() {
      return (
        <div>
          <HomeComponent />
        </div>
      )
   }
}

export default MasterComponent
```

Now second level component is named HomeComponent.js

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

class HomeComponent extends Component {
   render() {
      return (
        <div>
          <ProfileComponent/>
        </div>
      )
   }
}
export default HomeComponent
```

And third level Component named ProfileComponent in which we are going to consume context value

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

```
import { UserConsumer } from './UserContext';

class ProfileComponent extends Component {
   render() {
      return (
         <UserConsumer>
            {
               (username) => {
                  return <div>Hello {username}</div>
               }
            }
         </UserConsumer>
      )
   }
}
export default ProfileComponent
```

If in UserContext.js while creating context we will define default value and if in case we don't provide any value in component then it will take default value from UserContext,

```
//And App.js code will be as below,
import React from 'react';
import './App.css';
import './css/custom.css';
import MasterComponent from './components/MasterComponent'
import { UserProvider } from './components/UserContext';

function App() {
   return (
      <div className="App">
         {/* <UserProvider> */}
         <MasterComponent />
         {/* </UserProvider> */}
      </div>
   );
}
export default App;
```

While using context we can only pass one value, for multiple values we need to use multiple context.
There is one more way to access context value using Context Type property.

First we need to export UserContext from UserContext.js file,

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

```
const UserContext = React.createContext('Guest')
const UserProvider = UserContext.Provider
const UserConsumer = UserContext.Consumer

export {UserProvider,UserConsumer}
export default UserContext
```

Now we will consume context value using contextType in HomeComponent.js

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

class HomeComponent extends Component {
   render() {
      return (
         <div>
            Home Component {this.contextType}
            <ProfileComponent/>
         </div>
      )
   }
}

HomeComponent.contextType = UserContext

export default HomeComponent
```


Accessing multiple context values,

```
import React from 'react'
const UserContext = React.createContext('Guest')
const ThemeContext = React.createContext('theme')
const UserProvider = UserContext.Provider
const UserConsumer = UserContext.Consumer
const ThemeProvider = ThemeContext.Provider
const ThemeConsumer = ThemeContext.Consumer
export {UserProvider,UserConsumer,ThemeConsumer,ThemeProvider}

export default UserContext
```

and using it in App.js,

```
import React from 'react';
import './App.css';
import './css/custom.css';
```

```
import MasterComponent from './components/MasterComponent'
import { UserProvider, ThemeProvider } from './components/UserContext';

function App() {
   return (
      <div className="App">
         <UserProvider value="New User">
            <ThemeProvider value="red">
               <MasterComponent />
            </ThemeProvider>
         </UserProvider>
      </div>
   );
}

export default App;
```

Now consuming value of ThemeProvider in ProfileComponent.js file,

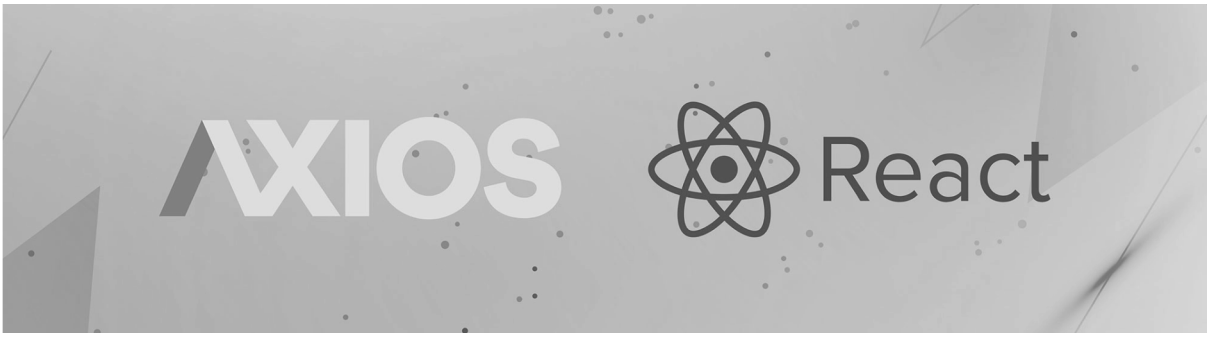
```
import React, { Component } from 'react'
import { UserConsumer,ThemeConsumer } from './UserContext';

class ProfileComponent extends Component {
   render() {
      return (
        <UserConsumer>
          {
             username => (
              <ThemeConsumer>
              {color => (
                  <div style={{ color: color }}>Hello {username}</div>
              )}
              </ThemeConsumer>
          )}
        </UserConsumer>
      )
   }
}
export default ProfileComponent
```

Limitation of Context

Context can only be used with the class component.

Using context type only a single context value can be accessed. For accessing multiple values, we will need to use nested consumer context.

HTTP



Basics of HTTP and React

So far, we have learned about the basics of React and how props and state are used. Now, we will learn how React makes API calls to fetch data and display in the browser. As React is just a library for the user interface, it only knows about Props and State so here, HTTP comes into the picture.

To fetch data in React, there are many HTTP libraries available like - **Apollo**, **Axios**, **Relay Modern**, **Request**, and **Superagent**. In this article, we are going to use Axios library.

To install axios

```
npm install axios
```

After the command is executed successfully, we will see in the package.json file that the dependencies are updated.

Now, we will see how we can fetch data and display it in the browser.

HTTP GET

Here, we are using an online API to fetch data from it instead of creating our own API. We are using **https://reqres.in/** to test our application.

Create a component named DisplayList.js.

```
import React, { Component } from 'react'
export class DisplayList extends Component {
  render() {
    return (
      <div>
        List of Users
</div>
    )
  }
}
export default DisplayList
```

And include this component in App.js.

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import DisplayList from './components/DisplayList'
function App() {
  return (
    <div className="App">
      <DisplayList></DisplayList>
    </div>
  );
}
export default App;
```

Now, we will fetch data from the library using Axios. For that, first, we need to import the axios library.

```
import axios from 'axios'
Now, to store the data that we get from API, we will create an empty array in state.
this.state = {
    Users:[]
}
```

Now, we will call the API and store the data in our array in componentDidMount() lifecycle method. This is executed when the component mounts for the first time and it is called only once.

So, the component will go like below.

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

const error ={
   color: 'red',
   fontWeight:'bold',
   fontSize:'14'
}

export class DisplayList extends Component {
   constructor(props) {
      super(props)

      this.state = {
         users:[],
          errorMessage:""
```

```
    }
  }

  componentDidMount(){
    axios.get('https://reqres.in/api/users/')
    .then(response => {
      console.log(response.data.data)
      this.setState({
        users:response.data.data
      })
    })
    .catch(error =>{
      this.setState({
        errorMessage:"Error fetching data"
      })
    })
  }
  render() {
    const {users,errorMessage} = this.state
    console.log(users.length)
    return (
      <div>
        List of Users
        {
          users.length?
          users.map(user=> <div key={user.id}><img src={user.avatar}
alt={user.first_name}/>{user.first_name + " "+ user.last_name}</div>):
          null
        }
        {errorMessage ? <div style={error}>{errorMessage}</div> : null}
      </div>
    )
  }
}

export default DisplayList
```

In the above, componentDidMount() will be called once the component is mounted, so it will display a list of users from the provided API. If the API is incorrect or does not return output, then the error message will be displayed.

HTTP and React

HTTP POST

Like HTTP Get, we use the Axios Post method to send the data to the API we use.

Example:

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

class UserForm extends Component {
  constructor(props) {
    super(props)

    this.state = {
      first_name: "",
      last_name: "",
      email: ""
    }
  }

  onChangeHandler = (e) => {
    this.setState({ [e.target.name]: e.target.value })
  }

  onSubmitHandler = (e) => {
    e.preventDefault();
    console.log(this.state)
    axios.post('https://reqres.in/api/users/',this.state)
    .then(response=>{
      console.log(response)
    })
    .catch(error => {
      console.log(error)
    })
  }

  render() {
    const { first_name, last_name, email } = this.state
    return (
      <div>
        <form onSubmit={this.onSubmitHandler}>
          <div>
            Email : <input type="text" name="email" value={email}
onChange={this.onChangeHandler} />
          </div>
          <div>
            First Name : <input type="text" name="first_name" value={first_name}
onChange={this.onChangeHandler} />
```

```
            </div>
            <div>
              Last Name : <input type="text" name="last_name" value={last_name}
onChange={this.onChangeHandler} />
            </div>
            <button type="submit">Submit</button>
          </form>
        </div>
      )
   }
}

export default UserForm
```

Let us include the UserForm component in App.js.

```
import React from 'react';
import './App.css';
import UserForm from './components/UserForm'

function App() {
  return (
    <div className="App">
      <UserForm></UserForm>
    </div>
  );
}

export default App;
```


We can also use the PUT, DELETE function in the same way.