React Hooks

useState:- useState allows functional components to have state, like *this.state* in class components. E.g. –

```
class CounterClass extends React.Component {
 constructor(props) {
  super(props);
  this.state = { count: 1 };
 }

 render() {
  return <div>
   <p>Count: {this.state.count}</p>
   <button onClick={() => this.setState({
    count: this.state.count + 1
   })}>Increase</button>
  </div>;
 }
}

function CounterFunction() {
 const [count, setCount] = React.useState(1);
 return (
  <div>
   <p>Count: {count}</p>
   <button onClick={() =>
    setCount(count + 1)}
   >Increase</button>
  </div>
 );
}

ReactDOM.render(
 <div>
  <CounterClass />
  <CounterFunction />
 </div>
, document.querySelector('#app'));
<script src="https://unpkg.com/react@16.7.0-alpha.0/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16.7.0-alpha.0/umd/react-dom.development.js"></script>

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

useEffect:- useEffect allows functional components to have lifecycle methods (such as componentDidMount, componentDidUpdate and componentWillUnmount) in one single API. E.g.-

```
class LifecycleClass extends React.Component {
 componentDidMount() {
  console.log('Mounted');
 }

 componentWillUnmount() {
  console.log('Will unmount');
 }

 render() {
```

```
   return <div>Lifecycle Class</div>;
  }
}

function LifecycleFunction() {
  React.useEffect(() => {
    console.log('Mounted');
    return () => {
      console.log('Will unmount');
    };
  }, []); // Empty array means to only run once on mount.
  return (
    <div>Lifecycle Function</div>
  );
}

ReactDOM.render(
  <div>
    <LifecycleClass />
    <LifecycleFunction />
  </div>
, document.querySelector('#app'));
<script src="https://unpkg.com/react@16.7.0-alpha.0/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16.7.0-alpha.0/umd/react-dom.development.js"></script>

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

useContext:- useContext hook makes it easy to pass data throughout your app without manually passing props down the tree. it is nothing but a global state to the app. It is a way to make a particular data available to all the components no matter how they are nested. Context helps you broadcast the data and changes happening to that data, to all the components.

To create a context in any React app, you need to follow 4 simple steps –

1- Create a context
2- Create a provider
3- Add provider to the app
4- UseContext

```
// UserDetailsProvider.js

import { createContext, useState } from "react";

//create a context, with createContext api
export const userDetailsContext = createContext();

const UserDetailsProvider = (props) => {
  // this state will be shared with all components
  const [userDetails, setUserDetails] = useState();

  return (
    // this is the provider providing state
    <userDetailsContext.Provider value={[userDetails, setUserDetails]}>
      {props.children}
    </userDetailsContext.Provider>
  );
};
```

```
export default UserDetailsProvider;
```

```
//App Component

import { BrowserRouter, Switch, Route } from "react-router-dom";
import { RouteWithSubRoutes } from "./utils/shared";
import UserDetailsProvider from "./context/UserDetailsProvider";
import routes from "./Routes";

function App() {
 return (
  <BrowserRouter>
   <Switch>
    // As login do not require the userDetails state, keeping it outside.
    <Route path="/" component={Login} exact />
    // All other routes are inside provider
    <UserDetailsProvider>
     {routes.map((route) => (
      <RouteWithSubRoutes key={route.key} {...route} />
     ))}
    </UserDetailsProvider>
   </Switch>
  </BrowserRouter>
 );
}

export default App;
```

```
// Profile.js

import { useEffect, useState, useContext } from "react";
import { getUser } from "../../utils/api";
import { userDetailsContext } from "../../context/UserDetailsProvider";

const Profile = ({ email }) => {
 // This is how we useContext!! Similar to useState
 const [userDetails, setUserDetails] = useContext(userDetailsContext);
 const [loading, setLoading] = useState(false);

 const handleGetUser = async () => {
  try {
   setLoading(true);
   let response = await getUser(email);
   setUserDetails(response.data);
  } catch (error) {
   console.log(error);
   // TODO: better error handling
  }
  setLoading(false);
 };

 useEffect(() => {
  if (!userDetails) {
```

```
    handleGetUser();
   }
 }, []);

 return <div className="bg-gray-gray1 h-full">// do something</div>;
};

export default Profile;
```

useReducer :- useReducer is usually preferable to useState when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. useReducer also lets you optimize performance for components that trigger deep updates because you can pass dispatch down instead of callbacks.

```
const { useReducer } = React;

function reducer(state, newState) {
 return {
  ...state,
  ...newState
 };
}

function App() {
 const [value, setValue] = useReducer(reducer, { a: "1" });


 function onSubmit() {
  setValue({ b: 2 });
 }
 return (
  <div className="App">
   <button onClick={onSubmit}>onSubmit</button>
   {JSON.stringify(value)}
  </div>
 );
}

ReactDOM.render(
 <App />,
 root
);
<script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>

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

useCallback:- useCallback() will return a ***memoized callback***. Normally, if you have a child component that receives a function prop, at each re-render of the parent component, this function will be re-executed; by using useCallback() you ensure that this function is only re-executed when any value on it's dependency array changes.

useCallback will return a memoized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders ***(e.g. shouldComponentUpdate)***.

```
function ExampleChild({ callbackFunction }) {
  const [value, setValue] = React.useState(0);

  React.useEffect(() => {
    setValue(value + 1)
  }, [callbackFunction]);

  return (<p>Child: {value}</p>);
}

function ExampleParent() {
  const [count, setCount] = React.useState(0);
  const [another, setAnother] = React.useState(0);

  const countCallback = React.useCallback(() => {
    return count;
  }, [count]);

  return (
    <div>
      <ExampleChild callbackFunction={countCallback} />
      <button onClick={() => setCount(count + 1)}>
        Change callback
      </button>

      <button onClick={() => setAnother(another + 1)}>
        Do not change callback
      </button>
    </div>
  )
}

ReactDOM.render(<ExampleParent />, document.getElementById('root'));
<script src="https://unpkg.com/react@16.8.0/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16.8.0/umd/react-dom.development.js"></script>

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

useMemo:- useMemo() will return a memoized value that is the result of the passed parameter. It means that useMemo() will make the calculation for some parameter once and it will then return the same result for the same parameter from a cache. This is very useful when you need to process a huge amount of data.

useMemo will only recompute the memoized value when one of the dependencies has changed. This optimization helps to avoid expensive calculations on every render.

1) A case of using memoization is React may be when you are trying to filter data from a large array.
2) Another case would be when you wish to transform a nested object based on some parameters into other object or array.

In such as case useMemo is really helpful. If the array and the filter criteria remains the same across re-renders, the calculation is not done again instead the previously calculated data is returned from the cache.

```
function ExampleChild({ value }) {
  const [childValue, setChildValue] = React.useState(0);

  React.useEffect(() => {
   setChildValue(childValue + 1);
  }, [value])

  return <p>Child value: {childValue}</p>;
}

function ExampleParent() {
 const [value, setValue] = React.useState(0);
 const heavyProcessing = () => {
  // Do some heavy processing with the parameter
  console.log(`Cached memo: ${value}`);
  return value;
 };

 const memoizedResult = React.useMemo(heavyProcessing, [value]);

 return (
  <div>
   <ExampleChild value={memoizedResult} />
   <button onClick={() => setValue(value + 1)}>
    Change memo
   </button>
  </div>
 )
}

ReactDOM.render(<ExampleParent />, document.getElementById('root'));
<script src="https://unpkg.com/react@16.8.0/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16.8.0/umd/react-dom.development.js"></script>

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

useRef:- useRef is a hook that uses the same ref throughout. It saves its value between re-renders in a functional component and doesn't create a new instance of the ref for every re-render. It persists the existing ref between re-renders.

The difference is that ***createRef*** will always create a new ref. In a class-based component, you would typically put the ref in an instance property during construction **(e.g. this.input = createRef())**. You don't have this option in a function component. ***useRef*** takes care of returning the same ref each time as on the initial rendering.

```
import React, { useRef, createRef, useState } from "react";
import ReactDOM from "react-dom";

function App() {
 const [renderIndex, setRenderIndex] = useState(1);
 const refFromUseRef = useRef();
 const refFromCreateRef = createRef();
 if (!refFromUseRef.current) {
  refFromUseRef.current = renderIndex;
 }
 if (!refFromCreateRef.current) {
  refFromCreateRef.current = renderIndex;
```

```
  }
  return (
    <div className="App">
      Current render index: {renderIndex}
      <br />
      First render index remembered within refFromUseRef.current:
      {refFromUseRef.current}
      <br />
      First render index unsuccessfully remembered within
      refFromCreateRef.current:
      {refFromCreateRef.current}
      <br />
      <button onClick={() => setRenderIndex(prev => prev + 1)}>
        Cause re-render
      </button>
    </div>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

useImperativeHandle:- Usually when you use useRef you are given the instance value of the component the ref is attached to. This allows you to interact with the DOM element directly. **useImperativeHandle** allows you to determine which properties will be exposed on a ref.

useImperativeHandle is very similar, but it lets you do two things:

1) It gives you control over the value that is returned. Instead of returning the instance element, you explicitly state what the return value will be (see snippet below).

2) It allows you to replace native functions (such as blur, focus, etc) with functions of your own, thus allowing side-effects to the normal behavior, or a different behavior altogether. Though, you can call the function whatever you like.

There could be many reasons you want might to do either of the above; you might not want to expose native properties to the parent or maybe you want to change the behavior of a native function. There could be many reasons. However, ***useImperativeHandle*** is rarely used.

useImperativeHandle customizes the instance value that is exposed to parent components when using ref.

Example

In this example, the value we'll get from the ref will only contain the function blur which we declared in our useImperativeHandle. It will not contain any other properties (I am logging the value to demonstrate this). The function itself is also "customized" to behave differently than what you'd normally expect. Here, it sets document.title and blurs the input when blur is invoked.

```
const MyInput = React.forwardRef((props, ref) => {
 const [val, setVal] = React.useState('');
 const inputRef = React.useRef();

 React.useImperativeHandle(ref, () => ({
  blur: () => {
   document.title = val;
   inputRef.current.blur();
  }
 }));

 return (
  <input
   ref={inputRef}
   val={val}
   onChange={e => setVal(e.target.value)}
   {...props}
  />
 );
});

const App = () => {
 const ref = React.useRef(null);
 const onBlur = () => {
  console.log(ref.current); // Only contains one property!
  ref.current.blur();
 };

 return <MyInput ref={ref} onBlur={onBlur} />;
};

ReactDOM.render(<App />, document.getElementById("app"));
<script
src="https://cdnjs.cloudflare.com/ajax/libs/react/16.8.1/umd/react.production.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/react-dom/16.8.1/umd/react-
dom.production.min.js"></script>
<div id="app"></div>
```

useLayoutEffect:- useLayoutEffect is the same as useEffect, but only fires once all DOM mutations are completed. Used in rare cases when you need to calculate the distance between elements after an update or do other post-update calculations / side-effects.

Example

Suppose you have a absolutely positioned element whose height might vary and you want to position another div beneath it. You could use **getBoundingCLientRect()** to calculate the parent's height and top properties and then just apply those to the top property of the child.

```
const Message = ({boxRef, children}) => {
 const msgRef = React.useRef(null);
 React.useLayoutEffect(() => {
  const rect = boxRef.current.getBoundingClientRect();
  msgRef.current.style.top = `${rect.height + rect.top}px`;
 }, []);

 return <span ref={msgRef} className="msg">{children}</span>;
};
```

```
const App = () => {
  const [show, setShow] = React.useState(false);
  const boxRef = React.useRef(null);

  return (
    <div>
      <div ref={boxRef} className="box" onClick={() => setShow(prev => !prev)}>Click me</div>
      {show && <Message boxRef={boxRef}>Foo bar baz</Message>}
    </div>
  );
};

ReactDOM.render(<App />, document.getElementById("app"));
.box {
  position: absolute;
  width: 100px;
  height: 100px;
  background: green;
  color: white;
}

.msg {
  position: relative;
  border: 1px solid red;
}
<script
src="https://cdnjs.cloudflare.com/ajax/libs/react/16.8.1/umd/react.production.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/react-dom/16.8.1/umd/react-
dom.production.min.js"></script>
<div id="app"></div>
```

useDebugValue:- Sometimes you might want to debug certain values or properties, but doing so might require expensive operations which might impact performance.

useDebugValue is only called when the React DevTools are open and the related hook is inspected, preventing any impact on performance.