React js Notes

What is React ?

ReactJS is a declarative, efficient, and flexible JavaScript library for building reusable UI components. It is an open-source, component-based front end library. A ReactJS application is made up of multiple components, each component responsible for outputting a small, reusable piece of HTML code. ReactJS uses virtual DOM based mechanism to fill data in HTML DOM. The virtual DOM works fast as it only changes individual DOM elements instead of reloading complete DOM every time.

What is 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 is a JavaScript Extension Syntax used in React to easily write HTML and JavaScript together.

React components.

Every application you will develop in React will be made up of pieces called components. Components in React basically return a piece of JSX code. A component is an independent, reusable code block which divides the UI into smaller pieces.

Components are two types

- Functional Components
- Class Components

Functional Components:

A functional component is basically a JavaScript/ES6 function that returns a React element (JSX).

```
function Welcome(props) {

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

}
```

So a Functional Component in React:

- is a JavaScript/ES6 function
- must return a React element (JSX)
- always starts with a capital letter (naming convention)
- takes props as a parameter if necessary

Class Components

The second type of component is the class component. Class components are ES6 classes that return JSX.

```
    class Welcome extends React.Component {

  render() {

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

  }

}
```

Different from functional components, class components must have an additional render( ) method for returning JSX.

Why Use Class Components?

We used to use class components because of "state". In the older versions of React (version < 16.8), it was not possible to use state inside functional components.

Therefore, we needed functional components for rendering UI only, whereas we'd use class components for data management and some additional operations (like life-cycle methods).

This has changed with the introduction of React Hooks, and now we can also use states in functional components as well.

A Class Component:

- is an ES6 class, will be a component once it ‘extends’ a React component.
- takes Props (in the constructor) if needed
- must have a render( ) method for returning JSX

What are Props in React?

React has a special object called a prop (stands for property) which we use to transport data from one component to another. React Props are like function arguments in JavaScript and attributes in HTML.

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


function Car(props) {

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

}
```

props only transport data in a one-way flow (only from parent to child components)

The Render Function

React's goal is in many ways to render HTML in a web page. React renders HTML to the web page by using a function called ReactDOM.render().

The ReactDOM.render() function takes two arguments, HTML code and an HTML element. The purpose of the function is to display the specified HTML code inside the specified HTML element.

React Hooks

Hooks were added to React in version 16.8.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

What is a Hook?

Hooks allow us to "hook" into React features such as state and lifecycle methods.State generally refers to application data or properties that need to be tracked.

There are 3 rules for hooks:

- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

React useState Hook

The React useState Hook allows us to track state in a function component. State generally refers to data or properties that need to be tracking in an application.

Initialize useState

We initialize our state by calling useState in our function component. useState accepts an initial state and returns two values:

- The current state.
- A function that updates the state.

React useEffect Hooks

- The useEffect Hook allows you to perform side effects in your components.
- Some examples of side effects are: fetching data, directly updating the DOM, and timers.
- useEffect accepts two arguments. The second argument is optional.
- useEffect(<function>, <dependency>)
- useEffect runs on every render.
- We should always include the second parameter which accepts an array. We can optionally pass dependencies to useEffect in this array.

Effect Cleanup

- Some effects require cleanup to reduce memory leaks.
- Timeouts, subscriptions, event listeners, and other effects that are no longer needed should be disposed.
- We do this by including a return function at the end of the useEffect Hook.

Only run the effect on the initial render

```
function Timer() {

  const [count, setCount] = useState(0);

  useEffect(() => {

    setTimeout(() => {

      setCount((count) => count + 1);

    }, 1000);

  }, []); // <- add empty brackets here

  return <h1>I've rendered {count} times!</h1>;

}
```

React Context

React Context is a way to manage state globally. It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.

Create Context

- To create context, you must Import createContext and initialize it:
- Next we'll use the Context Provider to wrap the tree of components that need the state Context.

Context Provider

- Wrap child components in the Context Provider and supply the state value.
- Now, all components in this tree will have access to the user Context.

Use the useContext Hook

- In order to use the Context in a child component, we need to access it using the useContext Hook.
- First, include the useContext in the import statement:

```
import { useState, createContext, useContext } from "react";

import ReactDOM from "react-dom/client";

const UserContext = createContext();

function Component1() {

  const [user, setUser] = useState("Jesse Hall");

  return (

    <UserContext.Provider value={user}>

      <h1>{`Hello ${user}!`}</h1>

      <Component2 />

    </UserContext.Provider>

  );

}

function Component2() {

  return (

    <>

      <h1>Component 2</h1>

      <Component3 />

    </>
  );
}
function Component3() {
  return (
    <>
      <h1>Component 3</h1>
      <Component4 />
    </>
  );
}
function Component4() {
  return (
    <>
      <h1>Component 4</h1>
      <Component5 />
    </>
  );
}
function Component5() {
  const user = useContext(UserContext);
  return (
    <>
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>

    </>

  );

}

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

root.render(<Component1 />);
```

React useRef Hook

- The useRef Hook allows you to persist values between renders.
- It can be used to store a mutable value that does not cause a re-render when updated.
- It can be used to access a DOM element directly.

Does Not Cause Re-renders

If we tried to count how many times our application renders using the useState Hook, we would be caught in an infinite loop since this Hook itself causes a re-render. To avoid this, we can use the useRef Hook.

Use useRef to track application renders.

```
import { useState, useEffect, useRef } from "react";

import ReactDOM from "react-dom/client";

function App() {

  const [inputValue, setInputValue] = useState("");

  const count = useRef(0);

  useEffect(() => {

    count.current = count.current + 1;

  });
  return (

    <>

      <input

        type="text"

        value={inputValue}

        onChange={(e) => setInputValue(e.target.value)}

      />

      <h1>Render Count: {count.current}</h1>

    </>

  );

}

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

root.render(<App />);
```

useRef() only returns one item. It returns an Object called current.

When we initialize useRef we set the initial value: useRef(0).

It's like doing this: const count = {current: 0}. We can access the count by using count.current

Accessing DOM Elements

- In general, we want to let React handle all DOM manipulation.
- But there are some instances where useRef can be used without causing issues.
- In React, we can add a ref attribute to an element to access it directly in the DOM.

Use useRef to focus the input:

```
import { useRef } from "react";

import ReactDOM from "react-dom/client";

function App() {

  const inputElement = useRef();

  const focusInput = () => {

    inputElement.current.focus();

  };

  return (

    <>

      <input type="text" ref={inputElement} />

      <button onClick={focusInput}>Focus Input</button>

    </>

  );

}

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

root.render(<App />);
```

Tracking State Changes

- The useRef Hook can also be used to keep track of previous state values.
- This is because we are able to persist useRef values between renders.

Use useRef to keep track of previous state values:

```
import { useState, useEffect, useRef } from "react";

import ReactDOM from "react-dom/client";

function App() {

  const [inputValue, setInputValue] = useState("");

  const previousInputValue = useRef("");

  useEffect(() => {

    previousInputValue.current = inputValue;

  }, [inputValue]);

  return (

    <>

      <input

        type="text"

        value={inputValue}

        onChange={(e) => setInputValue(e.target.value)}

      />

      <h2>Current Value: {inputValue}</h2>

      <h2>Previous Value: {previousInputValue.current}</h2>

    </>
  );

}

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

root.render(<App />);
```

React useReducer Hook

- The useReducer Hook is similar to the useState Hook.
- It allows for custom state logic.
- If you find yourself keeping track of multiple pieces of state that rely on complex logic, useReducer may be useful.
- The useReducer Hook accepts two arguments.
- useReducer(<reducer>, <initialState>)
- The reducer function contains your custom state logic and the initialStatecan be a simple value but generally will contain an object.
- The useReducer Hook returns the current stateand a dispatchmethod.

```
import { useReducer } from "react";

import ReactDOM from "react-dom/client";

const initialTodos = [

  {

    id: 1,

    title: "Todo 1",

    complete: false,

  },

  {

    id: 2,

    title: "Todo 2",

    complete: false,
},

];

const reducer = (state, action) => {

  switch (action.type) {

    case "COMPLETE":

      return state.map((todo) => {

        if (todo.id === action.id) {

          return { ...todo, complete: !todo.complete };

        } else {

          return todo;

        }

      });

    default:

      return state;

  }

};

function Todos() {

  const [todos, dispatch] = useReducer(reducer, initialTodos);

  const handleComplete = (todo) => {

    dispatch({ type: "COMPLETE", id: todo.id });

  };

  return (

    <>
```

```
      {todos.map((todo) => (
        <div key={todo.id}>
          <label>
            <input
              type="checkbox"
              checked={todo.complete}
              onChange={() => handleComplete(todo)}
            />
            {todo.title}
          </label>
        </div>
      ))}
    </>
  );
}

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

root.render(<Todos />);
```

React useCallback Hook

- The React useCallback Hook returns a memoized callback function.
- Think of memoization as caching a value so that it does not need to be recalculated.
- This allows us to isolate resource intensive functions so that they will not automatically run on every render.
- The useCallback Hook only runs when one of its dependencies update.
- This can improve performance.

    - The useCallback and useMemo Hooks are similar. The main difference is that useMemo returns a memoized value and useCallback returns a memoized function.

React useMemo Hook

- The React useMemo Hook returns a memoized value.
- Think of memoization as caching a value so that it does not need to be recalculated.
- The useMemo Hook only runs when one of its dependencies update.
- This can improve performance.
- The useMemo and useCallback Hooks are similar. The main difference is that useMemo returns a memoized value and useCallback returns a memoized function.