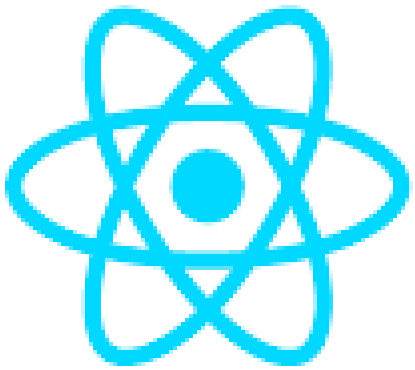

REACT NOTES

MUHAMMAD ZOHAIB IRSHAD
BSSE-F-24
Air University Islamabad

1. Installation of React through Vite:

First of all, you have to open the folder in vs code where you want to create a project. Then, open terminal and enter the following command:

```
    npm create vite@latest
```

Then enter the information that is required after that your first react app will be ready. Then you have to run following command after opening the created app in vs code.

```
    npm install
```

That's all and your react app is ready.

2. Components in React:

A component in React is a reusable piece of UI that can be used multiple times in an application. Components help break down complex UIs into smaller, manageable parts.

Basic Example:

```
  import React from "react";

const Greeting = () => {
  return <h1>Hello, Welcome to React!</h1>;
};

export default Greeting;
```

Usage in App.jsx file:

```
import Greeting from "./Greeting";

function App() {
  return (
    <div>
      <Greeting />
    </div>
  );
```

```
}

export default App;
```

3. Props in React:

Props **(**short for "properties"**)** allow components to receive data from their parent components. Props are read-only and help make components reusable**.**

Creating a component with prop:

```
const Greeting = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};

export default Greeting;
```

Using in the App.jsx file:

```
import Greeting from "./Greeting";

function App() {
  return (
    <div>
      <Greeting name="Zohaib" />
      <Greeting name="Ali" />
    </div>
  );
}

export default App;
```

Object destructuring concept for props:

```
const Greeting = ({ name }) => {
```

```
  return <h1>Hello, {name}!</h1>;
};
```

Using in App.jsx file:

```
import Greeting from "./Greeting";

function App() {
  return (
    <div>
      <Greeting name="Zohaib" />
      <Greeting name="Ali" />
    </div>
  );
}

export default App;
```

4. Hooks and States in React:

Introduction to React Hooks:

React Hooks are functions that allow you to use state and other React features in functional components. Introduced in React 16.8, Hooks eliminate the need for class components while maintaining the same functionality.

Why Use Hooks?

- Simplifies component logic.
- Reduces boilerplate code.
- Allows reuse of stateful logic across components.
- Enables better code organization.

useState:

The useState Hook allows you to add state to functional components.

Syntax:

```
import React, { useState } from ‘react’;

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count +
1)}>Increment</button>
      <button onClick={() => setCount(count –
1)}>Decrement</button>
    </div>
  );
}

export default Counter;
```

Explanation:

- useState(0): Initializes state with a default value of 0.
- count: The state variable.
- setCount: Function to update the state.
- State updates cause re-rendering of the component.

useEffect Hook:

The useEffect Hook is one of the most powerful and commonly used hooks in React. It allows you to perform **side effects** in functional components, such as:

- Fetching data
- Subscribing to events
- Updating the DOM
- Setting up timers

```
useEffect(() => {
  // Side effect logic here
}, [dependencies]);
```

- The **first argument** is a function where you write the side effect code.
- The **second argument** (optional) is a **dependency array** that determines when the effect should run.

Example of running useEffect on every render:

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

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

  useEffect(() => {
    console.log("Effect ran!");
  });

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count +
1)}>Increment</button>
    </div>
  );
};

export default Counter;
```

- Every time the component **re-renders**, the useEffect runs.
- Open the browser console, and you'll see "Effect ran!" logged **on every click**.

Example of state change in useEffect hook:

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

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

  useEffect(() => {
    document.title = `Count: ${count}`;
  }, [count]); // Runs when `count` changes
```

```
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count +
1)}>Increment</button>
    </div>
  );
};

export default WatchCount;
```

This updates the document title whenever **count** changes.

useRef Hook:

The useRef hook in React is used to reference a DOM element **or** persist values across renders without causing re-renders**.**

Syntax:

```
const refContainer = useRef(initialValue);
```

Example:

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

const FocusInput = () => {
  const inputRef = useRef(null); // Create a reference

  useEffect(() => {
    inputRef.current.focus(); // Automatically focus the
input field on mount
  }, []);

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Type
here..." />
      <button onClick={() => inputRef.current.focus()}>Focus
Input</button>
    </div>
```

```
  );
};

export default FocusInput;
```

- useRef(null) creates a reference and stores it in inputRef.
- The <input> element is linked to inputRef via ref={inputRef}.
- In useEffect, inputRef.current.focus() automatically focuses the input when the component loads.
- Clicking the "Focus Input" button will **re-focus** the input manually.

useContext Hook:

The useContext Hook allows components to access values from a React context without prop drilling.

Example:

```
import React, { createContext, useContext } from 'react';

const ThemeContext = createContext('light');

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button style={{ background: theme === 'dark' ?
'black' : 'white', color: theme === 'dark' ? 'white' :
'black' }}>Click Me</button>;
}

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <ThemedButton />
    </ThemeContext.Provider>
  );
}

export default App;
```

Explanation:

- createContext() creates a Context object.
- useContext(ThemeContext) accesses the context value directly.
- ThemeContext.Provider wraps the component tree to provide context values.

5. Conditional rendering in React:

Example:

```
import { useState } from 'react'
import './App.css'

function App() {
  const [first, setfirst] = useState(true)
  return (
    <>
      {first?<button>Show btn is
true</button>:<button>Showbtn is false</button>}
      <div className="card">
        <button onClick={() => setfirst(!first)}>
          Toggle Showbtn
        </button>
      </div>
    </>
  )
}

export default App
```

Explanation:

- Import useState from React and App.css for styling.
- Declare state variables: count (initialized to 0) and first (initialized to true).
- Render a button with conditional text based on the value of first.
- Create another button that toggles the value of first when clicked.
- Export the App component to be used elsewhere in the app.

6. List rendering in React:

Example:

```
import { useState } from 'react'
import './App.css'

function App() {
  const [todos, settodo] = useState([
    {
      title:"I am todo",
      desc:"I am desc"
    },
    {
      title:"I am todo 1",
      desc:"I am desc 1"
    },
    {
      title:"I am todo 2",
      desc:"I am desc 2"
    },
  ])
  const Todo=({todo})=>{
    return(
      <>
      <div className='todo'>{todo.title}</div>
      <div className='todo'>{todo.desc}</div>
      </>
    )
  }
     return (
    <>
      <div className="card">

        {todos.map(todo=>{
          return <Todo key={todo.title} todo={todo}/>
        }
        )}
      </div>
    </>
```

```
  )
}

export default App
```

Get Explanation from chatgpt.

Same example with different approach:

```
import { useState } from 'react'
import './App.css'

function App() {
  const [todos, settodo] = useState([
    {
      title:"I am todo",
      desc:"I am desc"
    },
    {
      title:"I am todo 1",
      desc:"I am desc 1"
    },
    {
      title:"I am todo 2",
      desc:"I am desc 2"
    },
  ])
     return (
    <>
      <div className="card">

        {todos.map(todo=>{
          return(
<>
<div key={todo.title}>
<div className='todo'>{todo.title}</div>
<div className='todo'>{todo.desc}</div>
</div>
```

```
</>
          )

        }
        )}
      </div>
    </>
  )
}

export default App
```

7. Handling Events in React:

In React, handling events is similar to handling events in regular HTML, but there are a few differences due to React's syntax and event handling system. Here's a simple guide to get you started.

Example:

```
import { useState } from 'react'
import './App.css'

function App() {

  const handleClick=()=>{
    alert("I am clicked.")
  }
  const handleMouse=()=>{
    alert("Mouse is over me.")
  }
  const [name, setname] = useState("Zohaib");
  const handleInput=(e)=>{
    setname(e.target.value)
  }
  return (
    <>
  <div className="container">
```

```
   <div> <button onClick={handleClick}>Click
me</button></div>
    <div><button onMouseOver={handleMouse}>Mouse
over</button></div>
    <div><input type="text" value={name} /></div>
  </div>
    </>
  )
}

export default App
```

This React code defines an App component with three interactive elements:

1. A button that shows an alert when clicked (handleClick).
2. A button that shows an alert when the mouse hovers over it (handleMouse).
3. An input field whose value is controlled by the name state, which can be updated (but lacks an onChange handler to do so).

The useState hook is used to manage the name state, which is initially set to "Zohaib". However, the input field doesn't currently allow the user to change the state because the handleInput function isn't connected to the onChange event of the input.

8. React Router: Routing in React:

Routing in React allows you to navigate between different components or pages without reloading the entire application. The most popular library for routing in React is **react-router-dom**.

You can install this library by running the following command in the terminal.

npm install react-router-dom

Setting Up React Router:

In your index.js or App.js, import BrowserRouter from react-router-dom and wrap your application with it:

```
import React from "react";
import ReactDOM from "react-dom";
import { BrowserRouter } from "react-router-dom";
import App from "./App";

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

Creating Routes:

In your App.js, set up the routes using Routes and Route components:

```
import React from "react";
import { Routes, Route } from "react-router-dom";
import Home from "./pages/Home";
import About from "./pages/About";
import Contact from "./pages/Contact";

const App = () => {
  return (
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
      <Route path="/contact" element={<Contact />} />
    </Routes>
  );
};

export default App;
```

Creating Page Componentfs:

Create separate component files for each page:

Home.js

```
import React from "react";
```

```
const Home = () => {
  return <h1>Home Page</h1>;
};

export default Home;
```

About.js

```
import React from "react";

const About = () => {
  return <h1>About Page</h1>;
};

export default About;
```

Contact.js

```
import React from "react";

const Contact = () => {
  return <h1>Contact Page</h1>;
};

export default Contact;
```

Adding Navigation:

To navigate between pages, use the `Link` component instead of `<a>`:

```
import React from "react";
import { Link } from "react-router-dom";

const Navbar = () => {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
```

```
      <Link to="/contact">Contact</Link>
    </nav>
  );
};

export default Navbar;
```

Include the `Navbar` component inside `App.js`:

```
import Navbar from "./components/Navbar";

const App = () => {
  return (
    <>
      <Navbar />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </>
  );
};
```

Using useNavigate for Programmatic Navigation:

```
import React from "react";
import { useNavigate } from "react-router-dom";

const Home = () => {
  const navigate = useNavigate();

  return (
    <div>
      <h1>Home Page</h1>
      <button onClick={() => navigate("/about")}>Go to
About</button>
    </div>
  );
};
```

```
export default Home;
```

Handling 404 Not Found Page:

If a user navigates to an undefined route, you can create a fallback route:

```
const NotFound = () => {
  return <h1>404 - Page Not Found</h1>;
};
```

Add this route to `App.js`:

```
<Route path="*" element={<NotFound />} />
```

9. useContext hook in React:

The useContext hook in React provides a way to share state and functions across components without having to pass props manually at every level. It is part of the React Context API and is useful for managing global state, such as theme settings, user authentication, or language preferences.

a) Understanding the useContext hook role in React:

When building applications, some data needs to be shared across multiple components. Without useContext, you might pass data as props (prop drilling), which can be cumbersome. Instead, React Context allows you to create a global state and access it from any component.

b) How to use useContext hook in React:

i. Create a Context
ii. Provide the Context value
iii. Consume the Context using useContext

c) Example: Counter App

Step 1: Create a Context.

Create a new folder in src with the name context and in this folder make a file named context.js. In this file, you should have following code:

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

export const counterContext=createContext(0);
```

Step 2: Provide the Context value in the app.

We will wrap up our code in the

```
<counterContext.Provider value={{count, setCount}}> </
counterContext.Provider>
```

So we can easily use the context api. So, our app.jsx file will be

```
import { useState } from 'react'
import Navbar from './components/Navbar'
import Button from './components/Button'
import component from './components/component'
import { counterContext } from './context/context'

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

  return (
    <>

   <counterContext.Provider value={{count, setCount}}>
<Navbar/>



      <div className="card">
        <button onClick={() => setCount((count) => count +
1)}>
          count is {count}
        </button>
       </div>
      </counterContext.Provider>
    </>
```

```
  )
}

export default App
```

Step 3: Consume the Context in Components.

So, we will use the count and setCount variable in the Navbar.jsx component by accessing through context api. See the code and understand it.

```
import React, { useContext } from 'react'
import Button from './Button'
import { counterContext } from '../context/context'

const Navbar = () => {
  const value=useContext(counterContext);
  return (
    <>
    <div>
      <h1>Navbar</h1>
      {value.count}
      <Button/>
    </div>
    </>
  )
}

export default Navbar
```

D) How useContext Works:

i. counterContext is created using createContext(0)
ii. <counterContext.Provider></counterContext.Provider> will wrap up the code and provides the initial state.
iii. **Now you can access count and setCount in every component by importing useContext (by importing through "import {counterContext} from '../context/context' statement). So, in this way, it can be updated by "const value=useContext(counterContext)" as written in the upper code.**

10. useMemo hook in React:

The useMemo hook in React is used for performance optimization. It memoizes (remembers) the result of a computation and recalculates it only when its dependencies change. This prevents unnecessary re-executions of expensive functions, making your app more efficient.

Syntax:

```
const memoizedValue = useMemo(() => computeFunction(),
[dependencies]);
```

i. computeFunction(): The function whose result we want to memoize.
ii. [dependencies]: The values that trigger recalculation when changed.

Example: Optimizing an Expensive Calculation:

Step 1: Basic Component Without useMemo

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

const Factorial = () => {
  const [number, setNumber] = useState(5);
  const [theme, setTheme] = useState(false);

  const factorial = (num) => {
    console.log("Calculating Factorial...");
    if (num <= 1) return 1;
    return num * factorial(num - 1);
  };

  return (
    <div style={{ background: theme ? "#333" : "#fff",
color: theme ? "#fff" : "#000", padding: "20px" }}>
      <h2>Factorial of {number}: {factorial(number)}</h2>
      <button onClick={() => setNumber(number + 1)}>Increase
Number</button>
```

```
      <button onClick={() => setTheme(!theme)}>Toggle
Theme</button>
    </div>
  );
};

export default Factorial;
```

Problems:

- The factorial() function runs every time the component re-renders.
- Changing the theme triggers the function unnecessarily.
- This can slow down performance if the function is complex

Step 2: Optimizing with useMemo:

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

const Factorial = () => {
  const [number, setNumber] = useState(5);
  const [theme, setTheme] = useState(false);

  const factorial = useMemo(() => {
    console.log("Calculating Factorial...");
    const computeFactorial = (num) => {
      if (num <= 1) return 1;
      return num * computeFactorial(num - 1);
    };
    return computeFactorial(number);
  }, [number]); // Runs only when `number` changes

  return (
    <div style={{ background: theme ? "#333" : "#fff",
color: theme ? "#fff" : "#000", padding: "20px" }}>
      <h2>Factorial of {number}: {factorial}</h2>
      <button onClick={() => setNumber(number + 1)}>Increase
Number</button>
      <button onClick={() => setTheme(!theme)}>Toggle
Theme</button>
```

```
    </div>
  );
};

export default Factorial;
```

Solution with useMemo:

- The factorial function only runs when number changes.
- Changing the theme will no longer re-run the factorial function.
- Performance is improved, especially for large computations.