React Hooks Documentation

1. Basic Hooks

1.1 useState

Must read this comme

- **Purpose**: To add state to functional components.
- **Syntax**:

  ```
  const [state, setState] = useState(initialValue);
  ```

- **Example**:

  ```
  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>
    </div>
   );
  }
  ```

1.2 useEffect

- **Purpose**: Perform side effects (e.g., fetching data, DOM manipulation, or subscriptions) in a functional component.
- **Syntax**:

```
useEffect(() => {
  // Side-effect code here
  return () => {
    // Cleanup function
  };
}, [dependencies]);
```

- **Example**:

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

function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds((prev) => prev + 1);
    }, 1000);

    return () => clearInterval(interval); // Cleanup on unmount
  }, []);

  return <div>Timer: {seconds}s</div>;
}
```

1.3 useContext

- **Purpose**: Access the value of React Context without a wrapper consumer.
- **Syntax**:

  ```
  const value = useContext(Context);
  ```

- **Example**:

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

  const ThemeContext = createContext("light");

  function ThemeComponent() {
   const theme = useContext(ThemeContext);
   return <div>Current Theme: {theme}</div>;
  }

  export default function App() {
   return (
    <ThemeContext.Provider value="dark">
     <ThemeComponent />
    </ThemeContext.Provider>
   );
  }
  ```

---

2. Additional Hooks

2.1 useReducer

- **Purpose**: Manages state with complex logic using reducers (like in Redux).

- **Syntax**:

  ```
  const [state, dispatch] = useReducer(reducer, initialState);
  ```

- **Example**:

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

  const reducer = (state, action) => {
    switch (action.type) {
      case "increment":
        return { count: state.count + 1 };
      case "decrement":
        return { count: state.count - 1 };
      default:
        return state;
    }
  };

  function Counter() {
    const [state, dispatch] = useReducer(reducer, { count: 0 });

    return (
      <div>
        <p>Count: {state.count}</p>
        <button onClick={() => dispatch({ type: "increment" })}>+</button>
        <button onClick={() => dispatch({ type: "decrement" })}>-</button>
  ```

```
    </div>
  );
}
```

2.2 useRef

- **Purpose**: Create a mutable reference that persists across renders. Often used to access DOM elements.
- **Syntax**:

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

- **Example**:

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

  function FocusInput() {
    const inputRef = useRef(null);

    const focusInput = () => {
      inputRef.current.focus();
    };

    return (
      <div>
        <input ref={inputRef} type="text" />
        <button onClick={focusInput}>Focus Input</button>
      </div>
    );
  ```

```
}
```

2.3 useMemo

- **Purpose**: Memoize expensive computations so they don't re-run unnecessarily.
- **Syntax**:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

- **Example**:

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

function ExpensiveComponent({ count }) {
  const expensiveValue = useMemo(() => {
    return count * 1000; // Expensive calculation
  }, [count]);

  return <div>Expensive Value: {expensiveValue}</div>;
}
```

2.4 useCallback

- **Purpose**: Memoize functions to prevent unnecessary re-renders.
- **Syntax**:

```
const memoizedCallback = useCallback(() => {
  // Callback function
}, [dependencies]);
```

- **Example**:

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

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

 const increment = useCallback(() => {
  setCount((prev) => prev + 1);
 }, []);

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

---

3. Special Hooks

3.1 useLayoutEffect

- **Purpose**: Like useEffect, but fires synchronously **after DOM mutations**.
- **Use Case**: Measure layout or DOM before the browser repaints.
- **Example**:

```
useLayoutEffect(() => {
```

```
  console.log("DOM is ready!");
}, []);
```

3.2 useImperativeHandle

- **Purpose**: Customizes the instance value of a React component when using forwardRef.
- **Syntax**:

```
useImperativeHandle(ref, () => ({ customMethod }));
```

- **Example**:

```
import React, { useRef, useImperativeHandle, forwardRef } from "react";

const Child = forwardRef((props, ref) => {
  useImperativeHandle(ref, () => ({
    sayHello() {
      alert("Hello from Child!");
    },
  }));

  return <div>Child Component</div>;
});

function Parent() {
  const childRef = useRef();

  return (
```

```
    <div>
      <Child ref={childRef} />
      <button onClick={() => childRef.current.sayHello()}>Say Hello</button>
    </div>
  );
}
```

3.3 useDebugValue

- **Purpose**: Adds a label for custom hooks when debugging.
- **Example**:

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

function useCustomHook(value) {
  const [state, setState] = useState(value);
  useDebugValue(state ? "ON" : "OFF");
  return [state, setState];
}
```

3.4 useId (React 18+)

- **Purpose**: Generate unique IDs for accessibility or form inputs.
- **Syntax**:

```
const id = useId();
```

- **Example**:

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

```
function MyInput() {
  const id = useId();
  return (
    <div>
      <label htmlFor={id}>Name:</label>
      <input id={id} />
    </div>
  );
}
```

Summary Table of Hooks:

Category	Hook	Purpose
State Management	useState	Adds state to functional components
Lifecycle/Side-Effects	useEffect	Perform side effects
Context API	useContext	Access context values
Reducers	useReducer	Manage state via reducers
Refs	useRef	Mutable DOM reference
Performance	useMemo, useCallback	Optimize performance
DOM Layout	useLayoutEffect	Synchronous layout effects
ForwardRefs	useImperativeHandle	Expose functions to parent refs

Category	Hook	Purpose
Debugging	useDebugValue	Custom hook debugging
ID Generation	useId	Unique ID for accessibility

React Router Hooks:

React Router provides several **hooks** to handle routing, navigation, and URL manipulation in React applications. These hooks are part of the **react-router-dom** library and allow programmatic interaction with routes.

---

1. useParams()

- **Purpose**: Retrieves dynamic parameters from the current route.
- **When to use**: When the route path has dynamic segments (e.g., /user/:id).

Example:

```
import { useParams } from 'react-router-dom';


function UserProfile() {
  const { id } = useParams();
  return <h1>User ID: {id}</h1>;
}
```

If the route is /user/123, the output will be:

User ID: 123

---

2. useNavigate()

- **Purpose**: Programmatically navigate between routes.

- **When to use**: For actions like redirecting on button click or after form submission.

Example:

```
import { useNavigate } from 'react-router-dom';

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

  const handleClick = () => {
    navigate('/about'); // Navigate to '/about'
  };

  return <button onClick={handleClick}>Go to About</button>;
}
```

- Additional options:
  - navigate(-1): Go back (like browser back).
  - navigate('/path', { replace: true }): Replace current entry in history.

---

3. useLocation()

- **Purpose**: Access details about the current URL.
- **When to use**: When you need access to pathname, search, or custom state.

Example:

```
import { useLocation } from 'react-router-dom';

function CurrentLocation() {
```

```
  const location = useLocation();

  return (
    <div>
      <h2>Current Path: {location.pathname}</h2>
      <p>Search Query: {location.search}</p>
    </div>
  );
}
```

If the URL is /about?name=John, it outputs:

Current Path: /about

Search Query: ?name=John

---

4. useSearchParams()

- **Purpose**: Manage query parameters in the URL.
- **When to use**: When working with search queries (e.g., ?key=value).

Example:

```
import { useSearchParams } from 'react-router-dom';

function SearchPage() {
  const [searchParams, setSearchParams] = useSearchParams();

  const handleChange = () => {
    setSearchParams({ name: 'John', age: 25 });
```

```
  };

  return (
    <div>
      <h1>Name: {searchParams.get('name')}</h1>
      <h2>Age: {searchParams.get('age')}</h2>
      <button onClick={handleChange}>Update Search Params</button>
    </div>
  );
}
```

If the current URL is /search, clicking the button changes it to:

/search?name=John&age=25

---

5. useMatch()

- **Purpose**: Checks if the current URL matches a specific path pattern.
- **When to use**: When conditionally rendering content based on route matching.

Example:

```
import { useMatch } from 'react-router-dom';

function MatchExample() {
  const match = useMatch('/about');

  return match ? <h1>You are on the About Page</h1> : <h1>Not About Page</h1>;
```

```
}
```

If the current route is /about, the output will be:

You are on the About Page

---

6. useOutlet()

- **Purpose**: Used in parent routes to render child routes dynamically.
- **When to use**: For nested routing.

Parent Route Example:

```
import { Outlet } from 'react-router-dom';

function ParentComponent() {
  return (
    <div>
      <h1>Parent Component</h1>
      <Outlet /> {/* Renders child routes */}
    </div>
  );
}
```

Child Route:

```
<Route path="/parent" element={<ParentComponent />}>
  <Route path="child" element={<h2>Child Component</h2>} />
</Route>
```

If the URL is /parent/child, the output will be:

Parent Component

Child Component

---

7. useRoutes()

- **Purpose**: Define routes declaratively using objects instead of JSX.
- **When to use**: When you want a programmatic approach to defining routes.

Example:

```
import { useRoutes } from 'react-router-dom';

function AppRoutes() {
  const routes = useRoutes([
    { path: '/', element: <Home /> },
    { path: '/about', element: <About /> },
  ]);

  return routes; // Renders matching route
}

function App() {
  return <AppRoutes />;
}
```

---

8. useNavigate() with state

- **Purpose**: Pass custom state while navigating.
- **When to use**: When sharing data during navigation without using query parameters.

Sender Component:

```
import { useNavigate } from 'react-router-dom';

function Sender() {
  const navigate = useNavigate();

  const sendData = () => {
    navigate('/receiver', { state: { message: 'Hello from Sender' } });
  };

  return <button onClick={sendData}>Send Data</button>;
}
```

Receiver Component:

```
import { useLocation } from 'react-router-dom';

function Receiver() {
  const location = useLocation();

  return <h1>Received: {location.state?.message}</h1>;
}
```

---

9. useOutletContext()

- **Purpose**: Share context between parent and child routes.
- **When to use**: When you need to pass data to nested routes.

Parent Component:

```
import { Outlet } from 'react-router-dom';

function ParentComponent() {
  const data = { user: 'John Doe' };

  return (
    <div>
      <h1>Parent Component</h1>
      <Outlet context={data} />
    </div>
  );
}
```

Child Component:

```
import { useOutletContext } from 'react-router-dom';

function ChildComponent() {
  const { user } = useOutletContext();

  return <h2>User: {user}</h2>;
}
```

Summary Table of React Router Hooks:

Hook	Purpose
useParams()	Access dynamic parameters from the route.
useNavigate()	Navigate programmatically between routes.
useLocation()	Get details about the current route.
useSearchParams()	Manage and update query parameters.
useMatch()	Check if the current route matches a path.
useOutlet()	Render nested child routes.
useRoutes()	Define routes using a JavaScript object.
useOutletContext()	Share context between parent and child routes.
