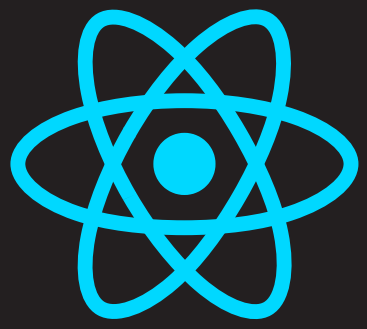
PORTFOLIO



The Hook Playbook

7 Fundamental Hooks of React

-> useState()

Brief: useState is a fundamental hook that lets you add state to functional components. It returns an array with two elements: the current state value and a function to update it.

Use Cases:

- **Managing form inputs**: Track and update user input in forms.
- **Toggle UI elements**: Control visibility of modal dialogs or dropdowns.

01

Code Example:

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

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

-> useEffect()

Use Cases:

- **Fetching data**: Load data from an API when the component mounts.
- **Setting up subscriptions**: Subscribe to WebSocket or other real-time data streams.
- **Updating the DOM**: Manually update the document title or other DOM elements.

03

Brief: useEffect allows you to perform side effects in your components, like fetching data, directly interacting with the DOM, or subscribing to observables.

Code Example:

```
function DataFetcher() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data));
  }, []);

  return <div>{data ? data.title : 'Loading...'}.
</div>;
}
```

04

-> useContext()

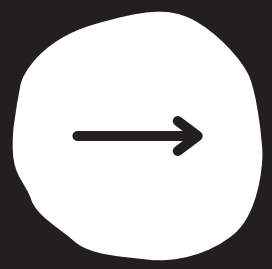
Brief: useContext allows you to consume a context value in your component without passing props down manually at every level.

Use Cases:

- **Theming**: Apply themes (dark/light mode) across your application.
- **Authentication**: Share user authentication status across multiple components.
- **Language switching**: Handle multiple languages or translations in a global context.

Code Example:

```
const ThemeContext = createContext('light');

function ThemedComponent() {
  const theme = useContext(ThemeContext);

  return <div className=
{`theme-${theme}`}>Current  Theme: {theme}
</div>;
}
```



-> useReducer()

Brief: useReducer is used for managing more complex state logic in your components, often for scenarios where useState would become cumbersome.

07

Use Cases:

- **Form management:** Handle complex form input logic.
- **State machines:** Implement state machines with clear transitions.
- **Complex state management:** Manage counters, toggles, and other states in a single reducer function.

CODE EXAMPLE:

```
const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

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

-> useRef()

Brief: useRef is a hook that returns a mutable ref object whose .current property can hold a value and persists across renders.

09

Use Cases:

- Accessing DOM elements: Directly interact with DOM elements, like focusing on input fields.
- Storing mutable values: Store values that don't trigger re-renders, like timers.
- Keeping track of previous values: Track the previous state without causing re-renders.

CODE EXAMPLE:

```
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>
  );
}
```

-> useMemo()

Brief: useMemo is a hook that memoizes a computed value to optimize performance by avoiding unnecessary recalculations on every render.

11

Use Cases:

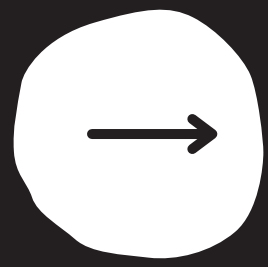
- **Expensive calculations:** Avoid recalculating intensive operations.
- **Optimizing rendering:** Prevent unnecessary re-renders of components relying on computed data.
- **Filtering large datasets:** Efficiently filter or process large datasets based on dependencies.

CODE EXAMPLE:

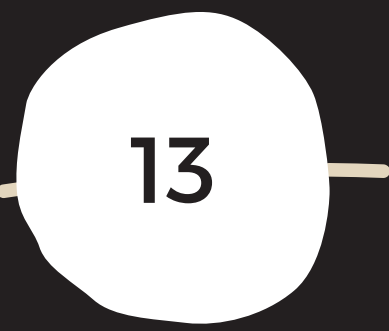
```
function ExpensiveComponent({ number }) {
 const [count, setCount] = useState(0);

 const expensiveCalculation = (num) => {
  console.log('Calculating...');
  return num * 2;
 };

 const doubledNumber = useMemo(() =>
expensiveCalculation(number), [number]);

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

12

-> useCallback()



Brief: useCallback is a hook that memoizes a function, ensuring that the same instance is used across renders unless dependencies change.



Use Cases:

- **Event handlers**: Memoize event handlers to prevent unnecessary re-renders of child components.
- **Optimizing component re-renders**: Avoid re-creating functions within components that rely on them.
- **Passing callbacks as props**: Prevent re-renders of components that rely on stable callback references.

CODE EXAMPLE:

14

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

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

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

Thanks Guys

Double Tap to Like