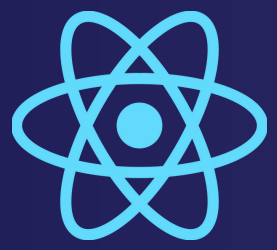
React Hooks Cheatsheet

@hassanbalouch_
@hassan-rind-3b8931241



React Hooks are built-in functions that allow you to use state and other react features, like lifecycle methods and context, in functional components without needing to write a class component


@hassanbalouch_


@hassan-rind-3b8931241

1. useState

This hook allows you to add state to functional components. The useState function returns a pair: the current state and a function that updates it.

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

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

  return (
    <div>
      <p>I am clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click Me!</button>
    </div>
  );
};
```

@hassanbalouch_
@hassan-rind-3b8931241

2. useEffect

This hook lets you perform side effects in functional components. It‘s a close replacement for componentDidMount, componentDidUpdate and componentWillUnmount in class components.

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

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;

    return () => {
      document.title = `React`;
    };
  }, [count]);

  return (
    <div>
      <p>I am clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click Me!</button>
    </div>
  );
};
```

@hassanbalouch_
@hassan-rind-3b8931241

3. useContext

This Hook lets you subscribe to React context without introducing nesting. It accepts a context object and returns the current context value for that context.

```jsx
import React, { useContext } from "react";
const ThemeContext = React.createContext("Light");

const ThemeButton = () => {
  const theme = useContext(ThemeContext);

  return (
    <div>
      <p>I am clicked {count} times</p>
      <button theme={theme}>I am styled by theme context1</button>
    </div>
  );
};
```

@hassanbalouch_
@hassan-rind-3b8931241

4. useReducer

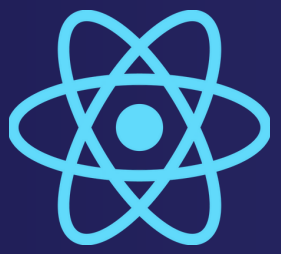
An alternative to useState. Accepts a reducer of type (state, action) => newState, and returns the current state paired with a dispatch method. Example:

It is particularly useful when the state logic is complex and involves multiple sub-values, or when the next state depends on the previous one. useReducer is an alternative to useState.

While useState is great for handling independent pieces of state, useReducer excels at handling more complex, interconnected state that involves multiple changes in an atomic and predictable manner.

@hassanbalouch_
@hassan-rind-3b8931241

4. useReducer



```jsx
import React, { useReducer } from "react";
const initialState = { count: 0 };

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

const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({ type: "decreament" })}>-</button>
      <button onClick={() => dispatch({ type: "increment" })}>+</button>
    </>
  );
};
```

@hassanbalouch_

@hassan-rind-3b8931241

5. useRef

This Hook creates a mutable ref object whose .current property is initialized to the passed argument. It‘s handy for keeping any mutable value around similar to how you‘d use instance fields in classes.

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

const TextInputWithFocusButton = () => {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    inputEl.current.focus();
  };

  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
};
```

@hassanbalouch_
@hassan-rind-3b8931241