React Hooks Cheatsheet

A quick reference for React Hooks, covering basic usage, rules, and common hooks with examples.

Basic Hooks

useState

Description:	Manages state within a functional component.
Usage:	`const [state, setState] = useState(initialState);`
Initial State:	Can be a value or a function that returns the initial value (lazy initialization).
setState:	A function to update the state. Triggers a re-render.

Example:	`import React, { useState } from 'react';`
	
	`function Example() {`
	`  const [count, setCount] = useState(0);`
	
	`  return (`
	`  <div>`
	`  <p>You clicked {count} times</p>`
	`  <button onClick={() => setCount(count + 1)}>`
	`  Click me`
	`  </button>`
	`  </div>`
	`  );`
	`}`

useEffect

Description:	Performs side effects in functional components (data fetching, subscriptions, etc.).

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

Dependencies:	An array of values. Effect runs only if these values change. `[]` means run only on mount/unmount. No array means run on every render (use with caution!).
Cleanup:	Optional function to clean up resources (e.g., unsubscribe from a subscription) when the component unmounts or before the effect runs again.

Example:	`import React, { useState, useEffect } from 'react';`
	
	`function Example() {`
	`  const [data, setData] = useState(null);`
	
	`  useEffect(() => {`
	`  async function fetchData() {`
	`  const response = await fetch('https://api.example.com/data');`
	`  const result = await response.json();`
	`  setData(result);`
	`  }`
	`  fetchData();`
	`  }, []);`
	
	`  return (`
	`  <div>`
	`  {data ? <p>Data: {JSON.stringify(data)}</p> : <p>Loading...</p>}`
	`  </div>`
	`  );`
	`}`

useContext

Description:	Consumes a context value. Allows accessing data from a provider higher up in the component tree.
Usage:	`const value = useContext(MyContext);`
Context:	Must be created using `React.createContext()`.
Provider:	A component that provides the context value to its children.

Example:	`import React, { createContext, useContext } from 'react';`
	
	`const ThemeContext = createContext('light');`
	
	`function ThemedButton() {`
	`  const theme = useContext(ThemeContext);`
	`  return <button style={{ backgroundColor: theme === 'dark' ? 'black' : 'white' }}>Themed Button</button>;`
	`}`
	
	`function App() {`
	`  return (`
	`  <ThemeContext.Provider value="dark">`
	`  <ThemedButton />`
	
	`</ThemeContext.Provider>`
	`  );`
	`}`

Additional Hooks

useRef

Description: Creates a mutable ref object whose `.current` property is initialized to the provided argument. Useful for accessing DOM elements or persisting values across renders without causing re-renders.

Usage:

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

Accessing Value: Access the current value via `myRef.current`.

Example:

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

function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` points to the mounted text input element

    inputEl.current.focus();
  };
  useEffect(() => {

    inputEl.current.focus();
  }, []);
  return (
    <>
      <input type="text" ref={inputEl} />
      <button onClick={onButtonClick}>
        Focus the input
      </button>
    </>
  );
}
```

useReducer

Description: An alternative to `useState`. Useful for complex state logic that involves multiple sub-values or when the next state depends on the previous one. Manages state transitions using a reducer function.

Usage:

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

Reducer: A function that takes the current state and an action, and returns the new state. `(state, action) => newState`

Dispatch: A function to dispatch actions to the reducer.

Example:

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

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

useMemo

Description: Memoizes a value. Only recomputes the value when one of the dependencies has changed. Improves performance by avoiding expensive calculations on every render.

Usage:

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

Dependencies: An array of dependencies. The function will only re-run when one of the dependencies changes.

Example:

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

function Example({ a, b }) {
  const memoizedValue = useMemo(() => {
    // Expensive calculation

    console.log('Calculating new value');
    return a + b;
  }, [a, b]);

  return (
    <div>
      Value: {memoizedValue}
    </div>
  );
}
```

Callback and Layout Hooks

useCallback

Description: Memoizes a callback function. Returns a memoized version of the callback that only changes if one of the dependencies has changed. Useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders.

Usage:

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

Dependencies: An array of dependencies. The callback will only be recreated when one of the dependencies changes.

Example:

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

function Example({ onClick }) {
  const handleClick = useCallback(() => {
    onClick('Button Clicked');
  }, [onClick]);

  return (
    <button onClick={handleClick}>Click
Me</button>
  );
}
```

useLayoutEffect

Description: Identical to `useEffect`, but it fires synchronously after all DOM mutations. Use this to read layout from the DOM and synchronously re-render. Prefer the standard `useEffect` when possible to avoid blocking visual updates.

Usage:

```
useLayoutEffect(() => {
  // DOM manipulation here
  return () => {
    // Cleanup
  }
}, [dependencies]);
```

When to Use: Use when you need to perform DOM measurements or mutations before the browser paints the screen to avoid visual glitches. Example: measuring the size of an element and adjusting its position.

Example:

```
import React, { useLayoutEffect, useRef, useState }
from 'react';

function Example() {
  const [height, setHeight] = useState(0);
  const elementRef = useRef(null);

  useLayoutEffect(() => {
    setHeight(elementRef.current.offsetHeight);
  }, []);

  return (
    <div ref={elementRef}>
      This element's height is: {height}px
    </div>
  );
}
```

Rules of Hooks

Key Principles

Only Call Hooks at the Top Level

Do not call Hooks inside loops, conditions, or nested functions. Ensure that Hooks are always called in the same order each time a component renders.

Only Call Hooks from React Functions

Call Hooks from React function components or custom Hooks. Do not call Hooks from regular JavaScript functions.

Linting Rules

The `eslint-plugin-react-hooks` ESLint plugin enforces these rules. Add it to your project for automatic checking.

```
npm install eslint-plugin-react-hooks --
save-dev
```

Then configure ESLint to use the plugin.

Custom Hooks

Description: Custom Hooks allow you to extract component logic into reusable functions. A custom Hook is a JavaScript function whose name starts with “use” and that may call other Hooks.

Example:

```
import { useState,
useEffect } from 'react';

function
useFriendStatus(friendID) {
  const [isOnline,
setIsOnline] =
useState(null);

  useEffect(() => {
    function
handleStatusChange(status)
{

setIsOnline(status.isOnline
);
    }

    //
ChatAPI.subscribeToFriendSt
atus(friendID,
handleStatusChange);
    setIsOnline(true);
    // Specify how to clean
up after this effect:
    return () => {
      //
ChatAPI.unsubscribeFromFrie
ndStatus(friendID,
handleStatusChange);
    };
  }, [friendID]);

  return isOnline;
}
```