React State CheatSheet

by Ndeye Fatou Diop

What is State?

- State is a built-in React object used to store data that can change over time.
- When state changes, the component re-renders to reflect the new state.

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

function Counter() {
  const [count, setCount] = useState(0);
  return <p>Current count: {count}</p>;
}
```

Initialising State

- State is initialised using the useState hook in functional components.
- The useState hook returns an array with the current state and a function to update it.

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

Using Previous State

When updating state based on the previous value, use a function inside setState.

```
const decrement = () => {
  setCount(prevCount =>
prevCount - 1);
}
```

Updating State

- Use the state update function (setState) to change the state.

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

  const increment = () =>
setCount(c => c + 1);

  return <button
onClick={increment}>Count:
{count}</button>;
}
```

Use an initialisation function to avoid re-running expensive calculations on every render.

```
// myExpensiveFunction is called only once, not on every re-
render
const [value, setValue] = useState(myExpensiveFunction);
```

Complex State (Objects and Arrays)

When updating state, always spread the existing state to avoid overwriting data.

```
const [user, setUser] = useState({ name: "Fatou", age: 30 });
const updateAge = () => setUser(user => ({ ...user, age: 25 }));
```

State Updates Are Asynchronous

- State updates are batched, which means multiple updates can be grouped and processed together.
- If you call setState multiple times in a row, React will batch them and only trigger one re-render.

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

  const handleClick = () => {
    setCount(count + 1);
    setCount(count + 1);
    setCount(count + 1);
    // Expected: 3, Actual: 1
(React batches updates)
  };

  return <button
onClick={handleClick}>Count:
{count}</button>;
}
```

Best Practices with State

- **Keep State Local:** Only lift state up if multiple components need to share it. This will improve the app performance since less components will render.
- **Use Reducers for Complex State Logic:** Use useReducer when you have multiple state variables that rely on complex interactions.
- **Avoid Unnecessary State:** Use props or derived data when possible.
- **Use refs when the state is not required for render:** Use refs to store mutable data, which is not required to render the UI (for example intervals)

Multiple State Variables

You can have multiple useState hooks to manage different pieces of state.

```
function UserProfile() {
  const [name, setName] = useState("Fatou");
  const [age, setAge] = useState(30);
  return <p>{name} is {age} years old.</p>;
}
```

State vs Props

- State is local to the component and can change over time.
- Props are read-only data passed from a parent component to a child component.

```
function ParentComponent({ initialCount })
{
  const [count, setCount] =
useState(initialCount);
  return <ChildComponent count={count} />;
}
```