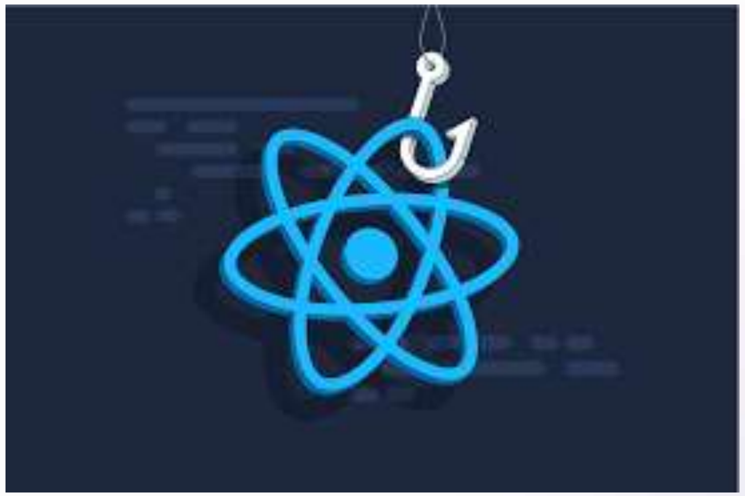
sourcemind

REACT JS

FUNCTIONAL COMPONENTS

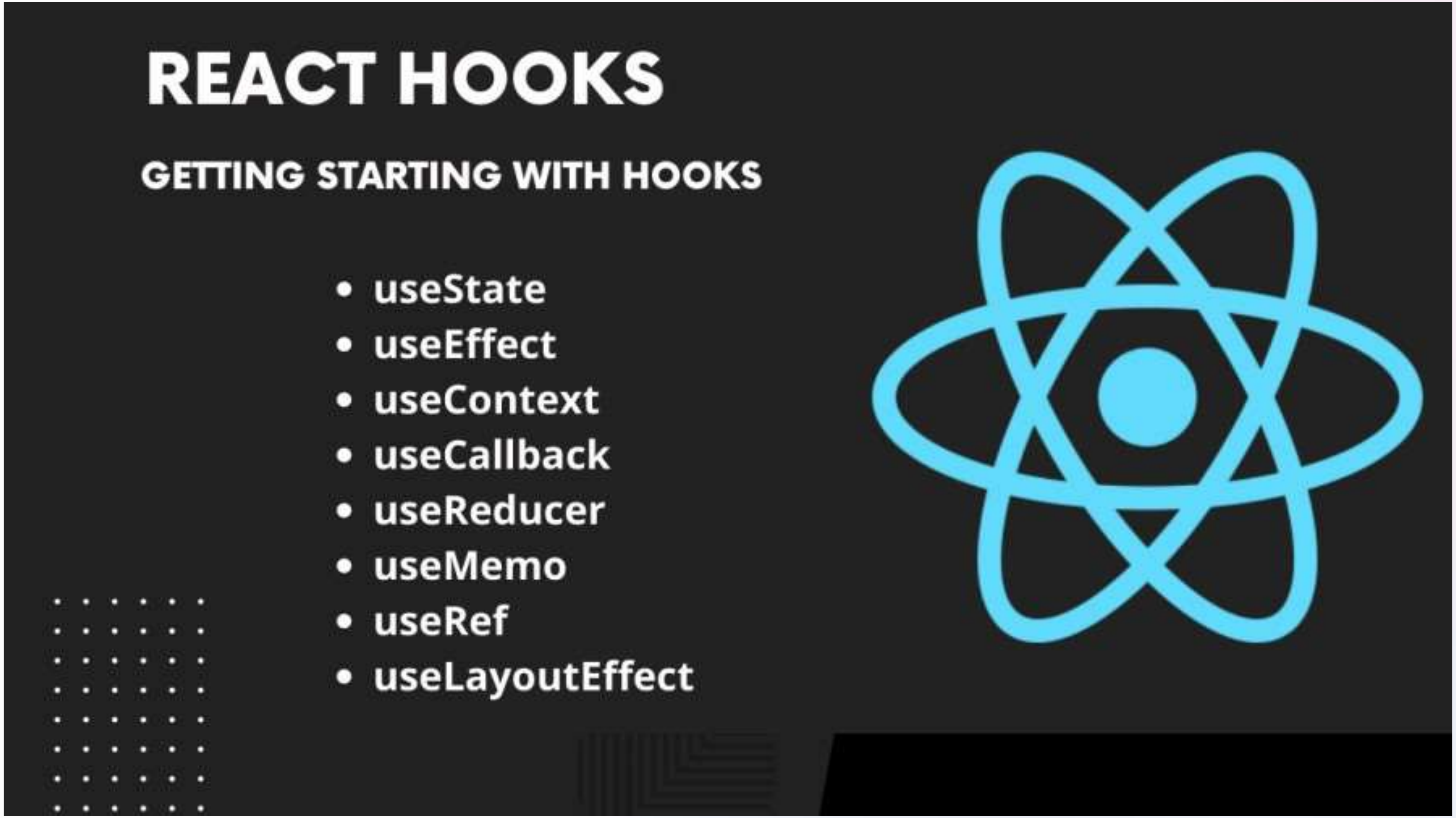
React JS

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.



Hooks allow function components to have access to state and other React features.

Because of this, class components are generally no longer needed.



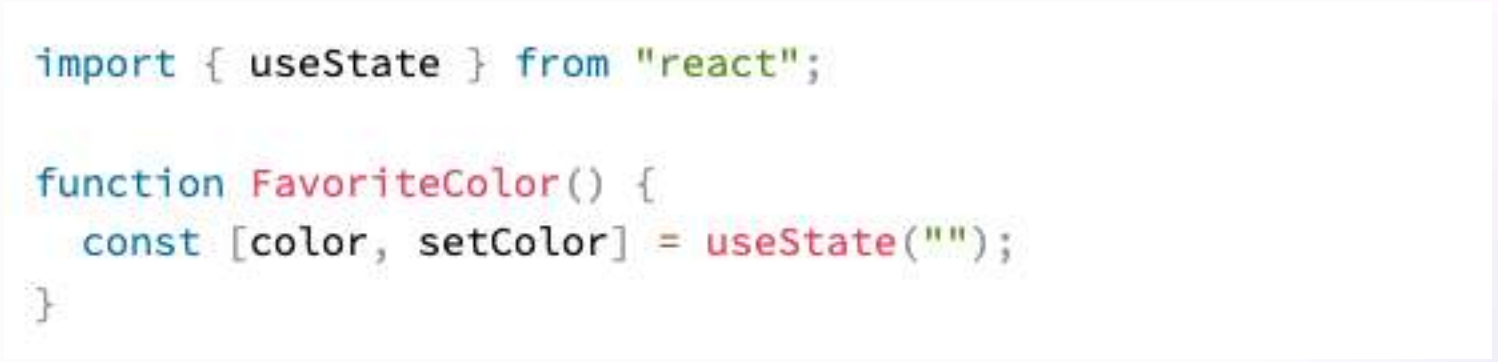
REACT HOOKS

GETTING STARTING WITH HOOKS

- useState
- useEffect
- useContext
- useCallback
- useReducer
- useMemo
- useRef
- useLayoutEffect

useState

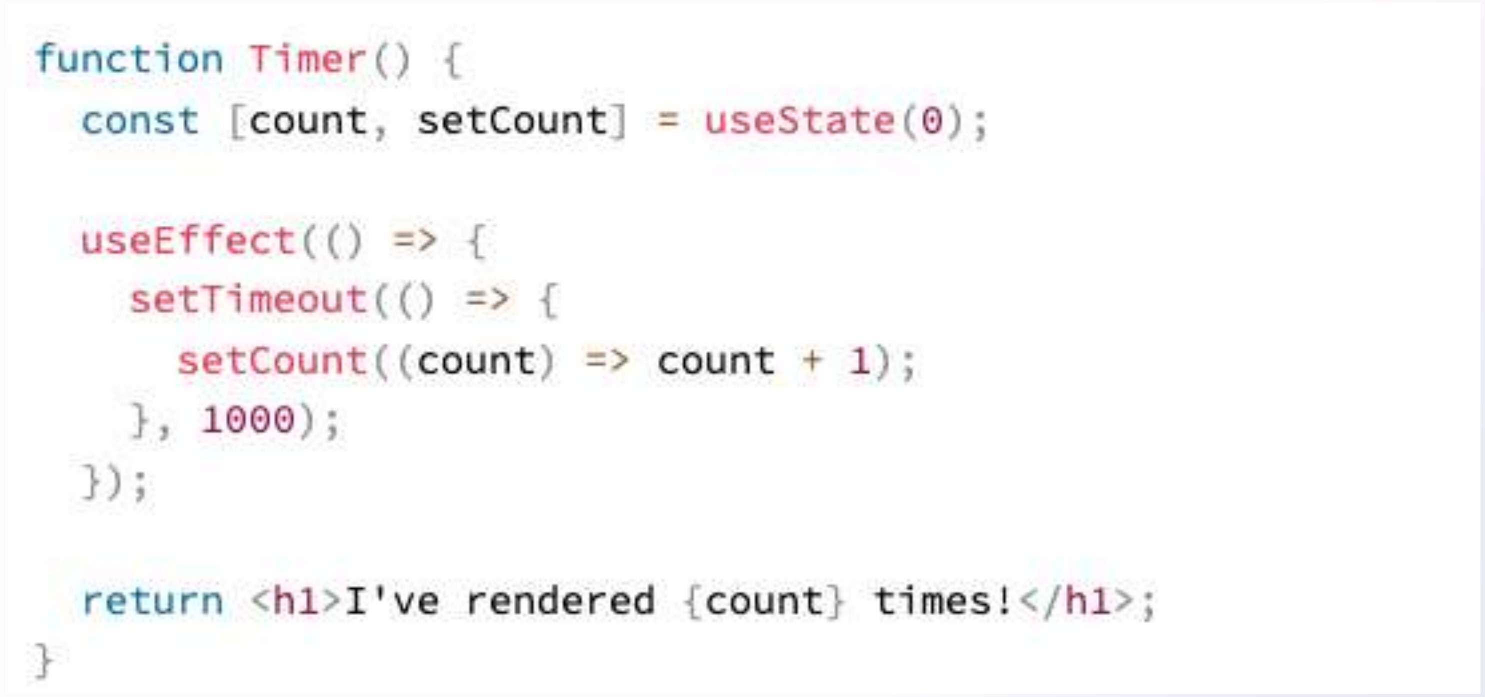
The React useState Hook allows us to track state in a function component.



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

function FavoriteColor() {
  const [color, setColor] = useState("");
}
```

useEffect

The useEffect Hook allows you to perform side effects in your components.
Some examples of side effects are: fetching data, directly updating the DOM, and timers.
useEffect accepts two arguments. The second argument is optional.
useEffect(<function>, <dependency>)

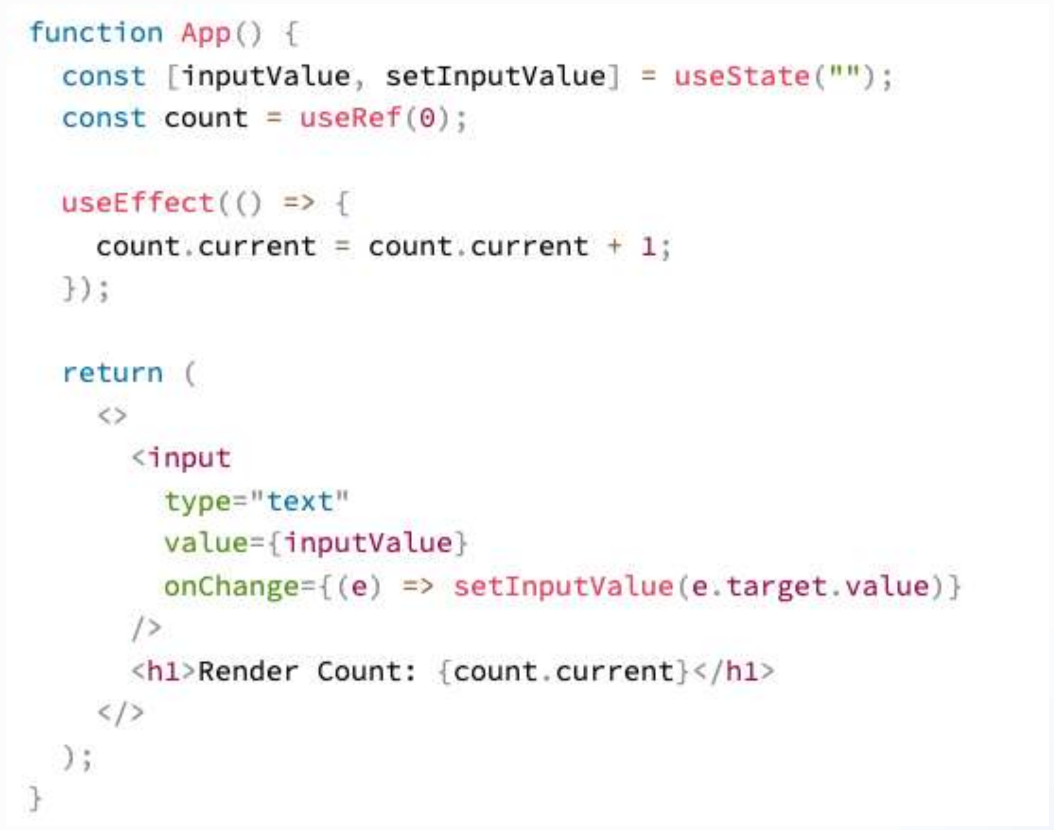


```
function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  });

  return <h1>I've rendered {count} times!</h1>;
}
```

useRef

The useRef Hook allows you to persist values between renders.

It can be used to store a mutable value that does not cause a re-render when updated.
It can be used to access a DOM element directly.

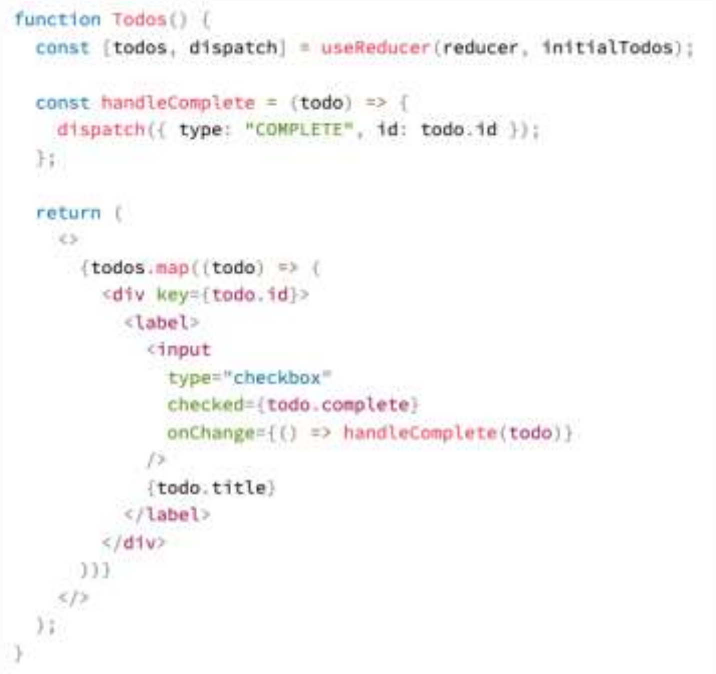


```
function App() {
  const [inputValue, setInputValue] = useState("");
  const count = useRef(0);

  useEffect(() => {
    count.current = count.current + 1;
  });

  return (
    <>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
      />
      <h1>Render Count: {count.current}</h1>
    </>
  );
}
```

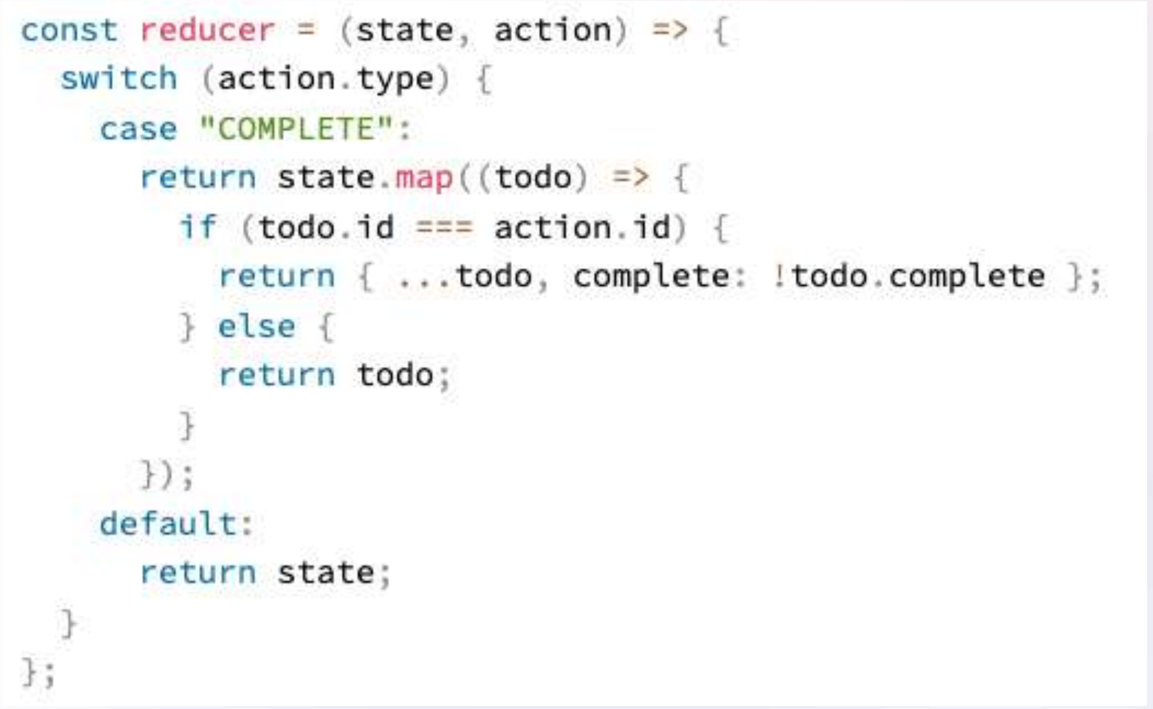
useReducer

The useReducer Hook is similar to the useState Hook.

It allows for custom state logic.
If you find yourself keeping track of multiple pieces of state that rely on complex logic, useReducer may be useful.



```
function Todos() {
  const [todos, dispatch] = useReducer(reducer, initialTodos);

  const handleComplete = (todo) => {
    dispatch({ type: "COMPLETE", id: todo.id });
  };

  return (
    <>
      {todos.map((todo) => (
        <div key={todo.id}>
          <label>
            <input
              type="checkbox"
              checked={todo.complete}
              onChange={() => handleComplete(todo)}
            />
            {todo.title}
          </label>
        </div>
      ))}
    </>
  );
}
```



```
const reducer = (state, action) => {
  switch (action.type) {
    case "COMPLETE":
      return state.map((todo) => {
        if (todo.id === action.id) {
          return { ...todo, complete: !todo.complete };
        } else {
          return todo;
        }
      });
    default:
      return state;
  }
};
```

sourcemind

useCallBack

The React useCallback Hook returns a memoized callback function.

useMemo

The React useMemo Hook returns a memoized value.

Custom Hooks

Hooks are reusable functions.

When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook.
Custom Hooks start with "use". Example: useFetch

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

const useFetch = (url) => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(url)
      .then((res) => res.json())
      .then((data) => setData(data));
  }, [url]);

  return [data];
};

export default useFetch;
```

```
import ReactDOM from "react-dom/client";
import useFetch from "./useFetch";

const Home = () => {
  const [data] = useFetch("https://jsonplaceholder.typicode.com/todos");

  return (
    <>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })}
    </>
  );
};
```

sourcemind

SASS

Syntactically Awesome Style Sheets

Sass is a stylesheet language that’s compiled to CSS.
It allows you to use variables, nested rules, mixins, functions, and more, all with a fully CSS-compatible syntax.

Sass helps keep large stylesheets well-organized and makes it easy to share design within and across projects.