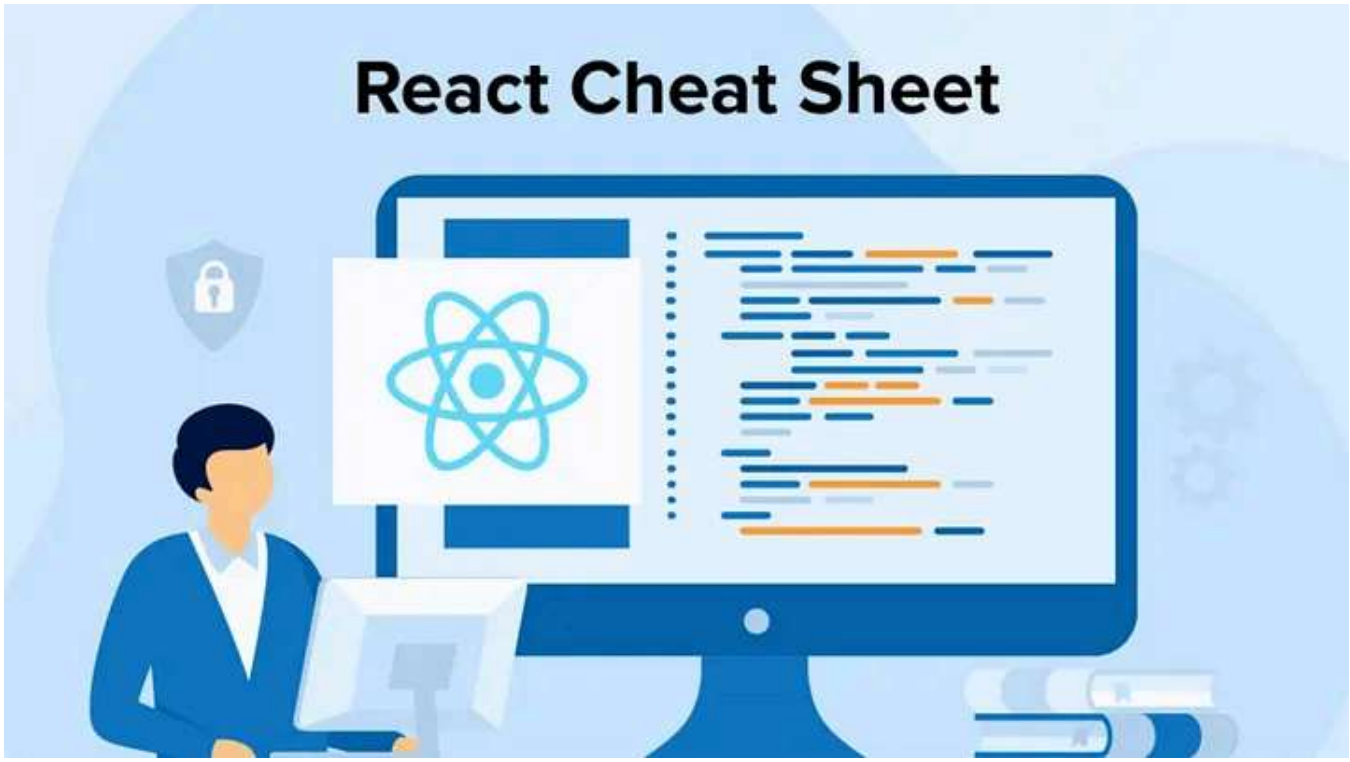
A Complete React Hooks Cheat Sheet for 2025 🎯📖

🚀 Why do you need the React Hooks cheat sheet? In the React world, the introduction to Hooks was a 🏠 welcome addition by the developer community. Essentially. 🚋

The Expert Developer · Follow
14 min read · Mar 29, 2025

Listen Share

> Hooks are functions that let you “hook into” React state and lifecycle features from functional components. ☎️



They are backward-compatible and don’t work with classes. If you are someone who is new to the world of Hooks or even a seasoned pro, this cheat sheet will help you to get all of your doubts and queries clear.

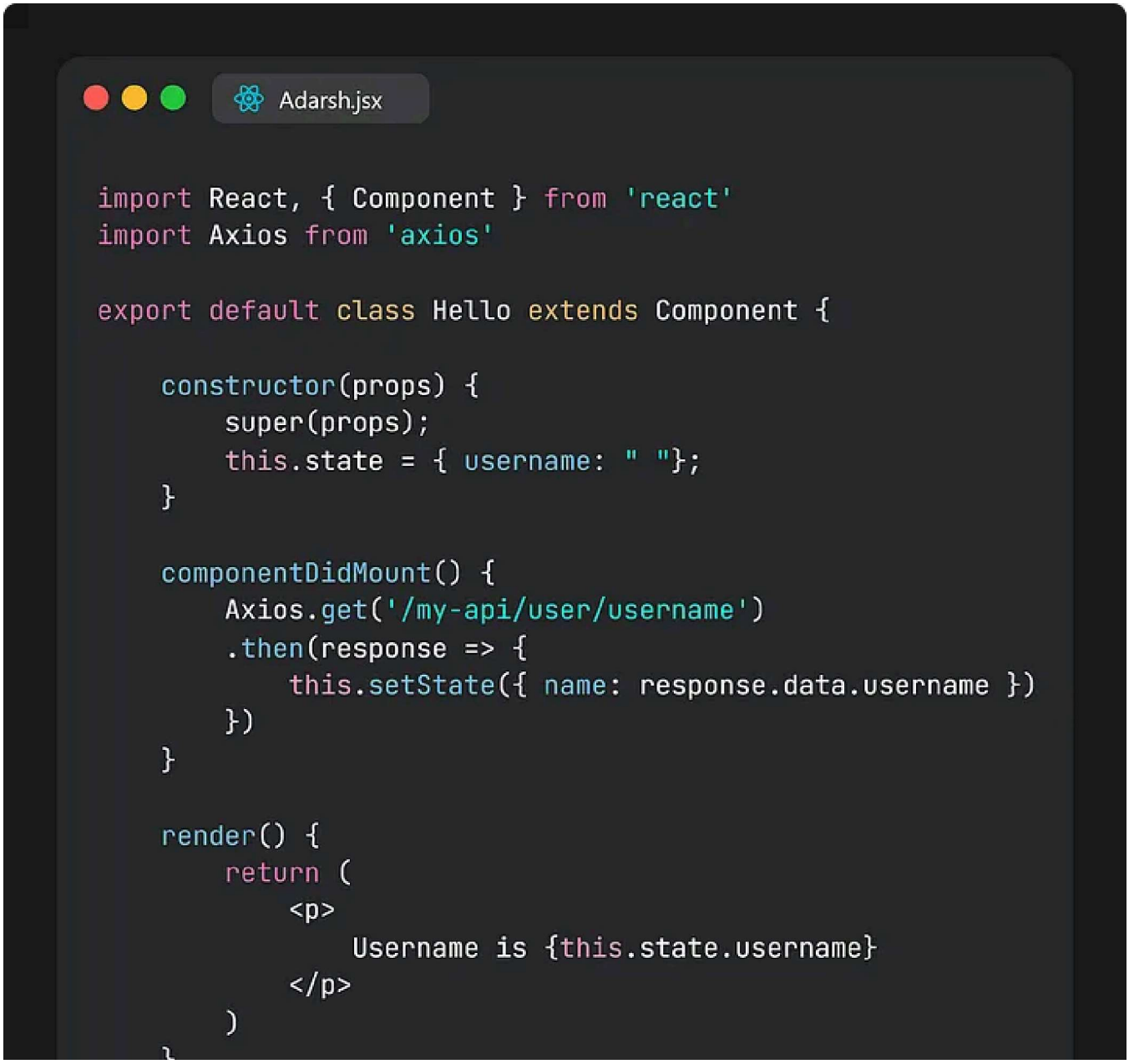
Basic level — React Hooks Cheat Sheet

1. Why were Hooks introduced, or why do we need them?

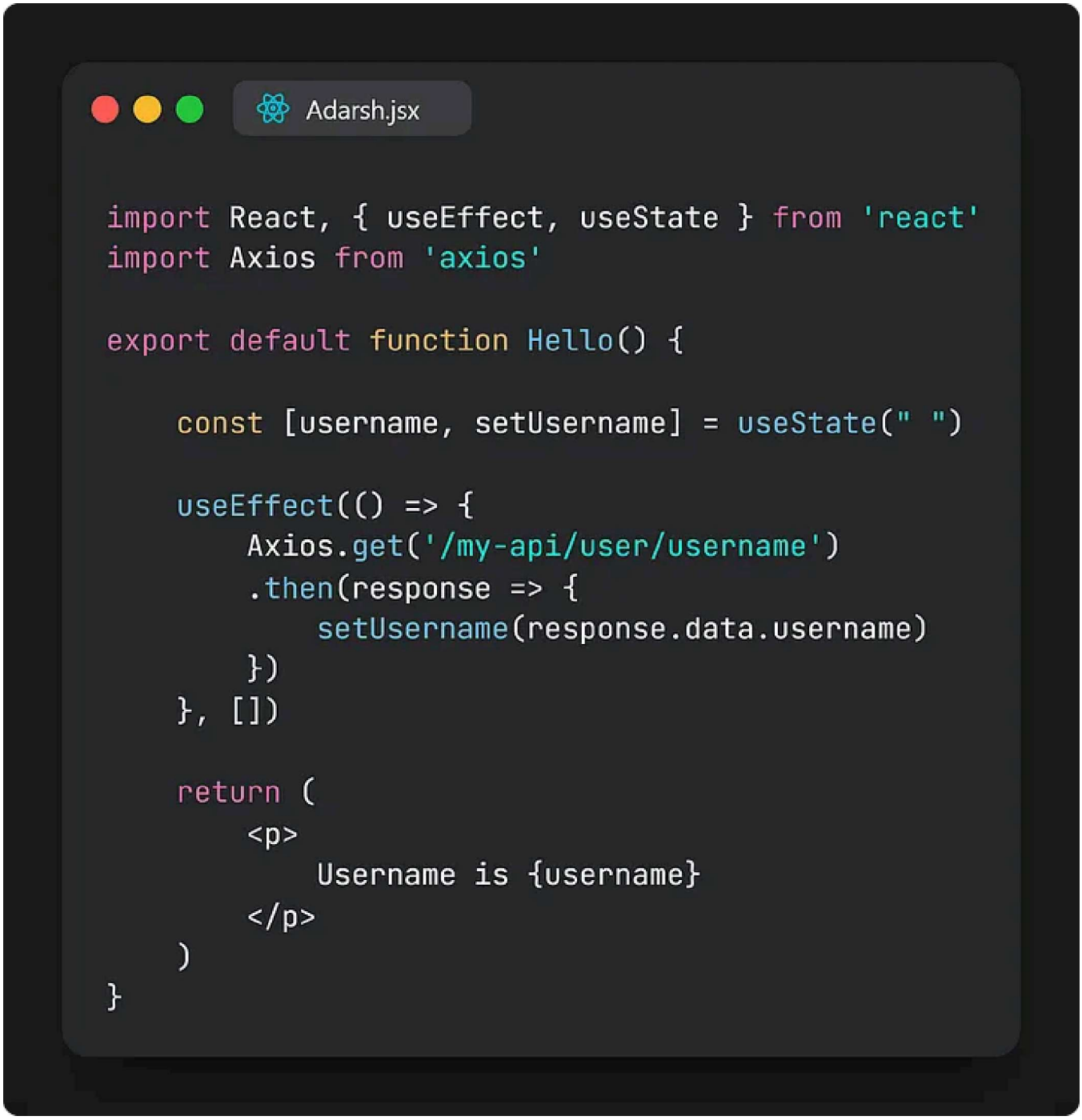
The React team introduced Hooks for the following reasons:

1. To reuse stateful logic without changing the component hierarchy: with Hooks, we can extract stateful logic from a component so that it can be tested independently and reused when needed. This makes it easy to share Hooks among many components or with someone else on the team.
   Patterns, like render props and Higher Order Components (HOCs), require you to restructure your components. This can be cumbersome and makes the code harder to follow. Sharing stateful logic became better after Hooks were introduced.

2. To split one component into smaller functions: Hooks have also made coding lifecycle methods easier than ever. Each of its lifecycle methods often contains a mix of unrelated logic. Components might perform data fetching using `componentDidMount()`, on the other hand, this method might include some unrelated logic that sets up event listeners.



While if we want to do the same with the built-in React Hooks like `useState` and `useEffect` we will do it like:

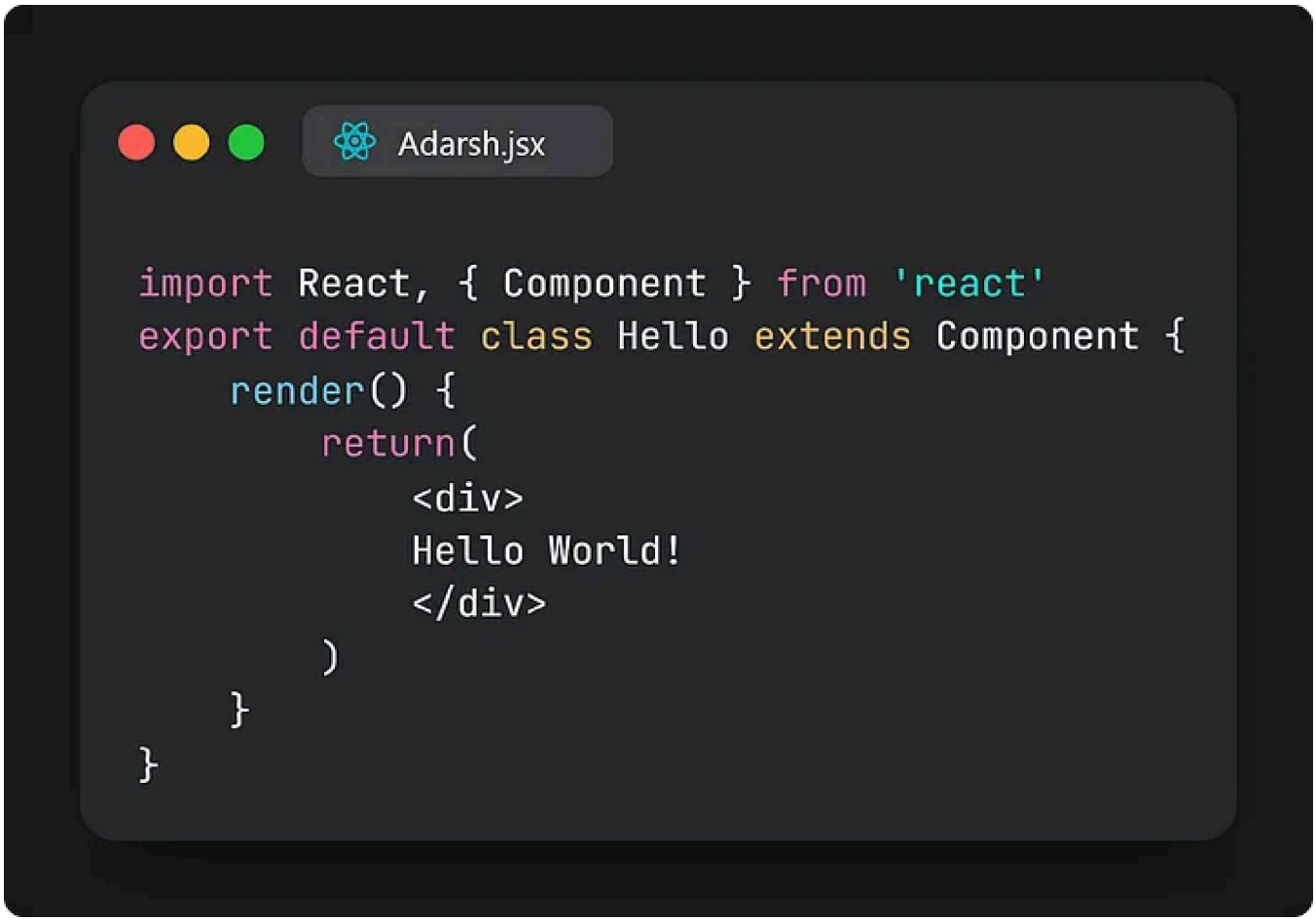


The code written using these two Hooks is more straightforward and concise. Moreover, it's easy to understand and work on if we work in a team. Using lifecycle methods like `componentDidMount()`, `componentDidUpdate()`, and `componentWillUnmount()`, they are handled individually, but when using React Hooks, you can handle all of this with just the `useEffect()` Hook.

3. To use more React features without classes: prior to when there was no concept of Hooks in React, its components used mainly class-based components. Making your components with classes all the time required great (and sometimes unnecessary) effort because you always had to switch between the classes, render props, HOCs,

etc. Thanks to React Hooks, now you can do all of these without switching using functional components.

- Using class-based components:



- Using functional components:

If you compare both of these, the second approach to using a functional component is a significantly simpler code that gets us the same result. You don't need to allocate

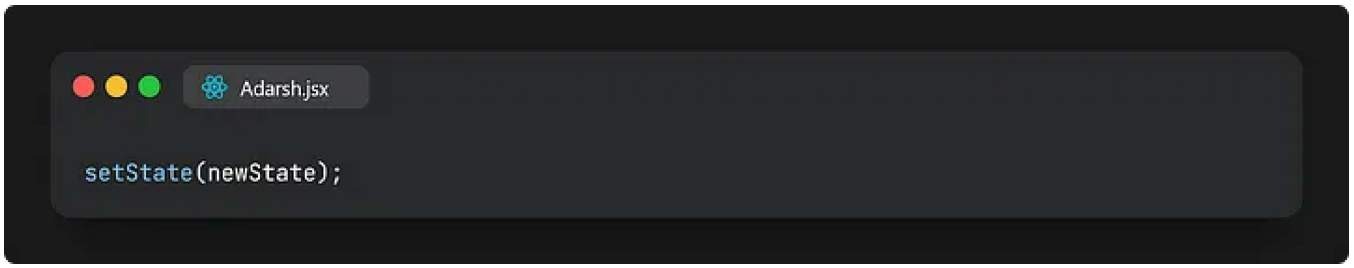
space to a class instance and then call the `render()` function. Instead, you simply call the function.

2. The `useState` Hook

> The `useState` Hook lets you use the local app state within a functional component.

It returns a stateful value along with a function to update it. Here's its basic call signature:

The `setState` function is used to update the state, and it accepts a new state value (`newState`):



Here are some of the crucial points regarding `useState` :

1. Declaring a state variable: to declare a state variable, you just need to call `useState` with some initial value:

2. Updating a state variable: here, you need to invoke the update function returned by invoking `useState` :

Adarsh.jsx

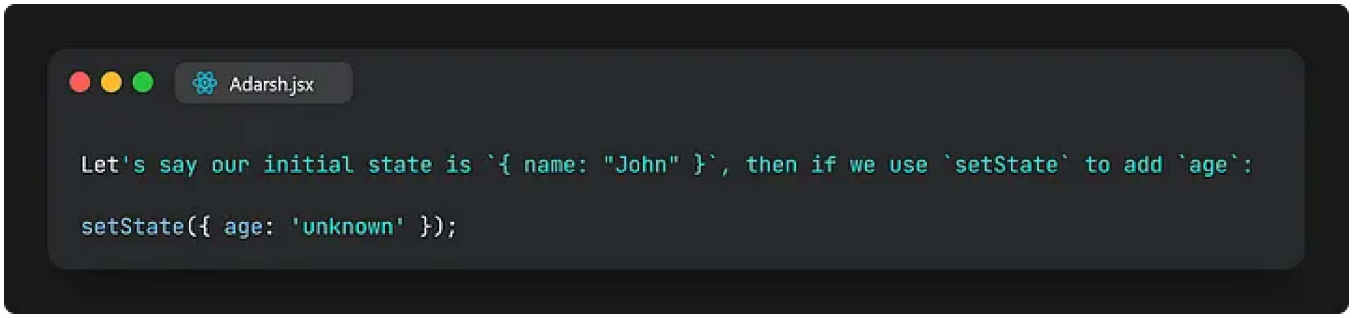
```
() => {
  const [items, setItems] = useState(0)
  const handleClick = () => setItems(items + 1)

return (
    <p>
      The user has {items} number of items.
      <p>
        <button onClick={handleClick}>Increase number of items by 1</button>
      </div>
    </div>
  )
}
```

3. Using multi-state variables: you can use multiple-state variables and update them within a functional component, as shown in this example:

Adarsh.jsx

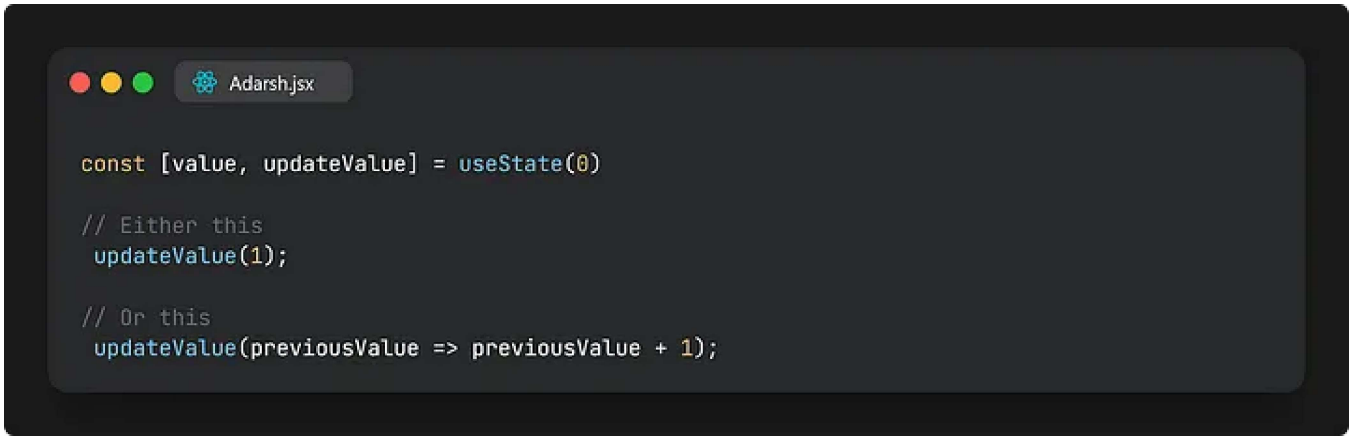
```
() => {
  const [items, setItems] = useState(0)
  const [price, setPrice] =
    useState(10)

  const handleItems = () => setItems(items + 1)
  const handlePrice = () =>
      setPrice(price + 10)

  return (
    <div>
      <p>The user has {items} number of items.</p>
      <p>And they sell items worth {price} USD.</p>

      <div>
        <button onClick={handleItems}>
          Increase number of items by 1
        </button>
        <button onClick={handlePrice}>
           Increase price by 10
        </button>
      </div>
    </div>
  )
}
```

4. Using object state variables: you can also use an entire object as an initial value passed to `useState`. It will not automatically merge update objects.



The new state object will become. `{ name: "John", age: "unknown" }`.

5. Functional `useState`: the updater function returned after invoking `useState` can also take a function just like what we used to do in class-based component's `setState`:



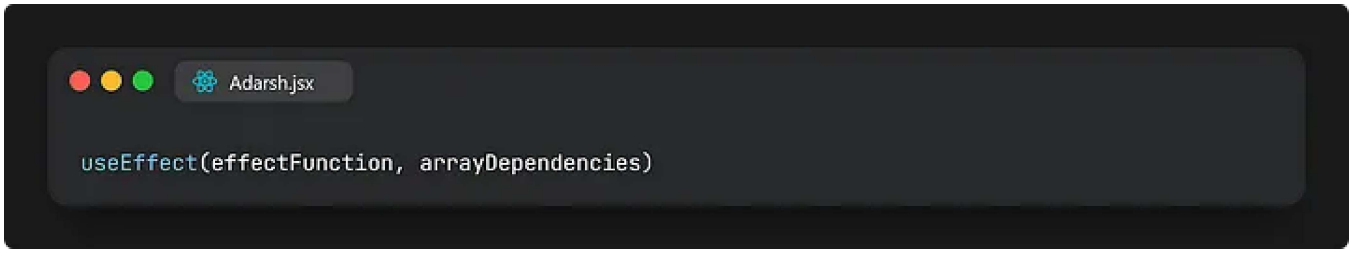
Both these forms are valid for invoking `updateValue`.

3. The `useEffect` Hook

> The `useEffect` Hook accepts a function that is used to perform any side effects.

The function passed to `useEffect` will run *after* the render is committed to the screen. These effects run after the render is complete, but we can also choose to fire them based on changes in certain values.

Here's the call signature of `useEffect`:



To make a basic side effect, we will be using both `useState` and `useEffect` Hooks in the following example:

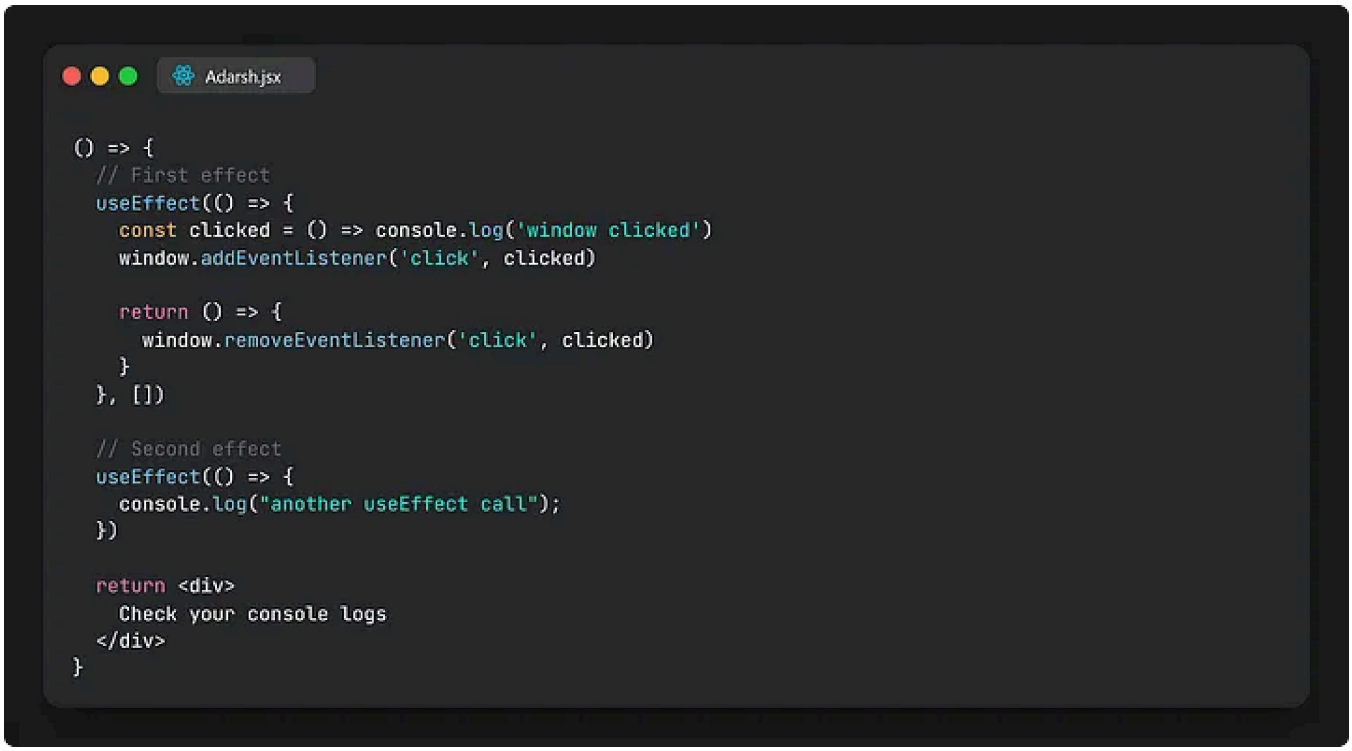
The `useEffect` block updates the current tab/browser window's title after running the `handleClick` function.

The following points are helpful regarding `useEffect`:

1. Cleaning up an effect: we must clean up the effect after a period, and this is usually done by returning a function from within the effect function passed to `useEffect`.

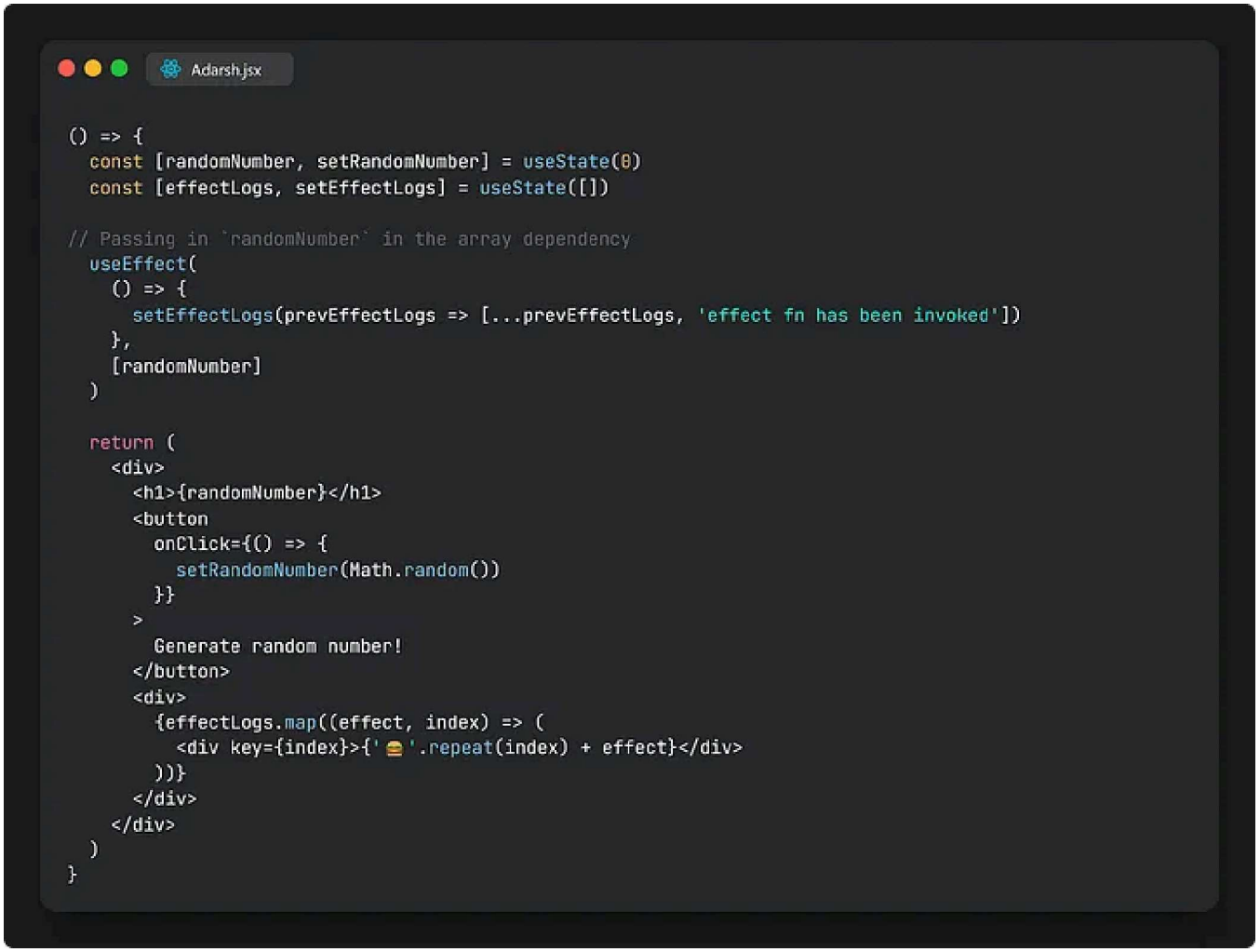
This cleanup function is run to prevent memory leaks before the component is removed from the UI. Also, if we write a new cleanup, the previous effect is cleaned up before executing the next one.

2. Creating multiple effects: to add multiple effects, we can have more than one `useEffect` calls within a functional component, as shown:

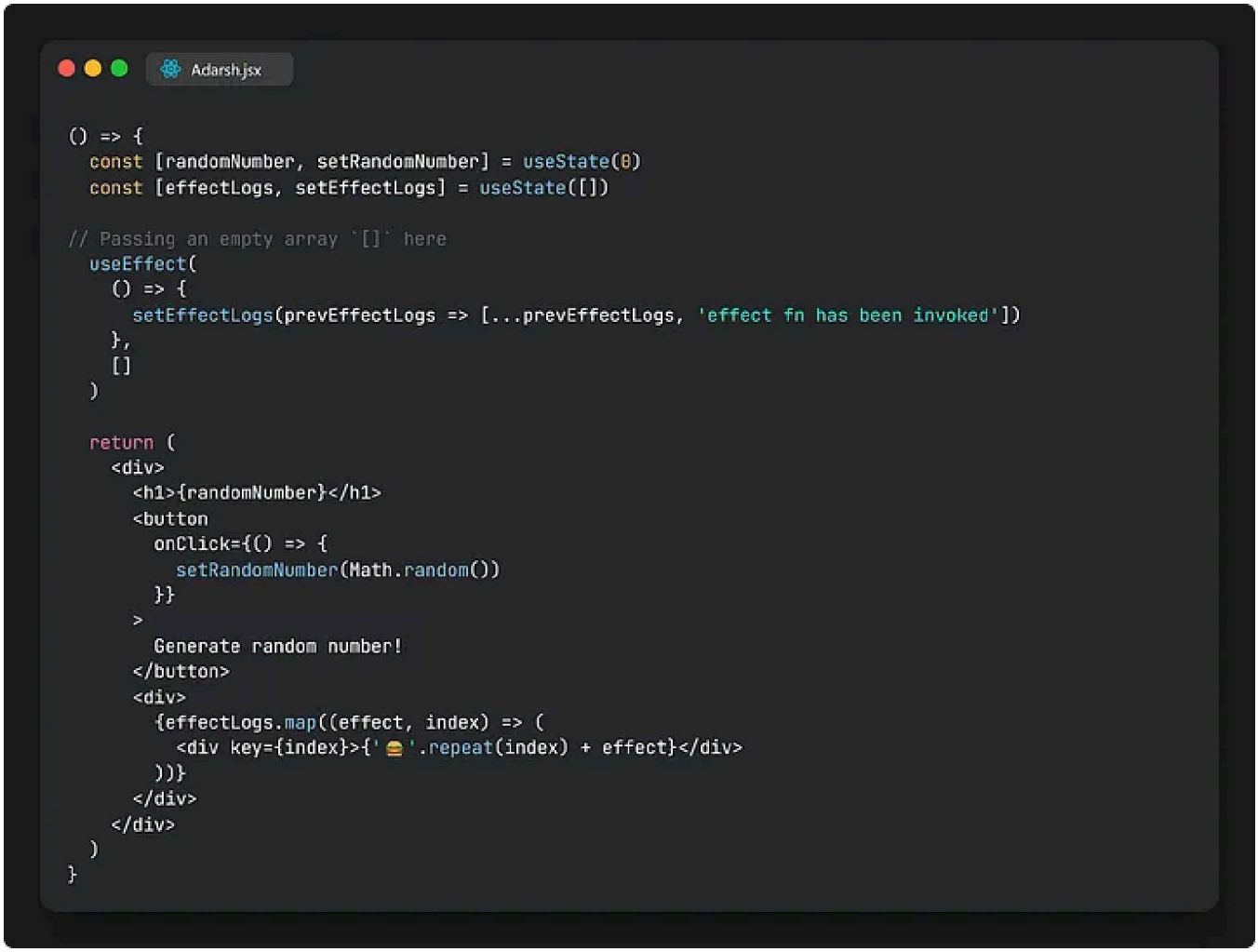


3. Multiple ways to skip effects: we can skip `useEffect` calls not to invoke it on every render. This can be done in multiple ways:

- **Using an array dependency**: here, `useEffect` is passed an array of a value. With this, the effect function will be called while mounting *and* whenever that value we passed is generated. Here's an example:



- **Using an empty array dependency:** in this case, `useEffect` is passed an empty array `[]`. The effect function is now called only while mounting:



- **Using no array dependency**: we can entirely skip effects without providing any array dependency. The effect function will be run after every single render:

4. Rules of Hooks

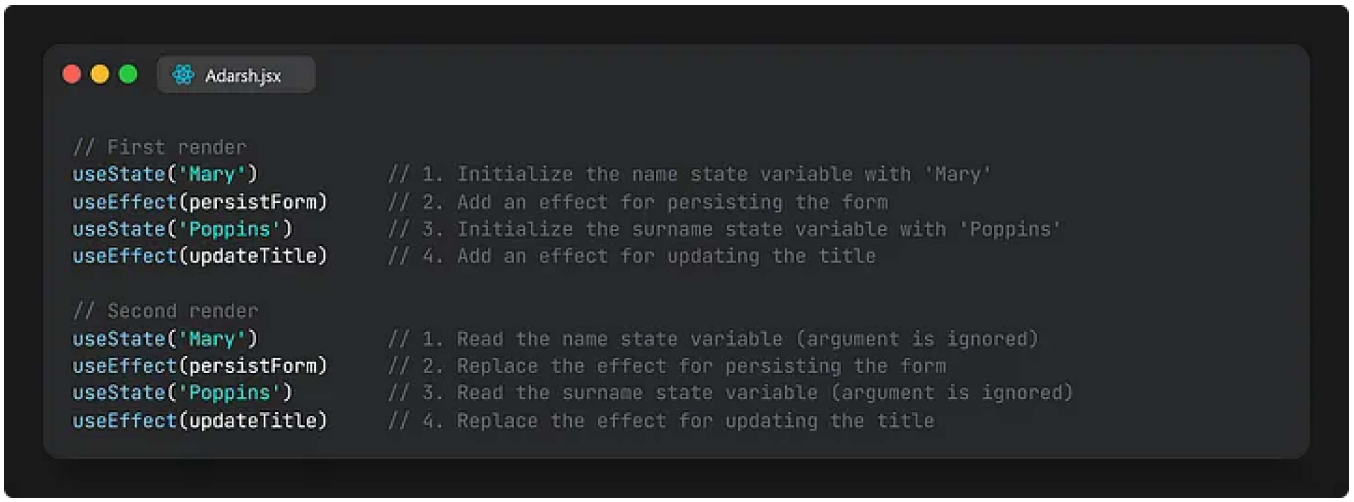
To effectively use React Hooks, we need to follow two basic rules when using them:

1. Call Hooks at the Top Level: this rule states that:

Don’t call Hooks inside loops, conditions, or nested functions.

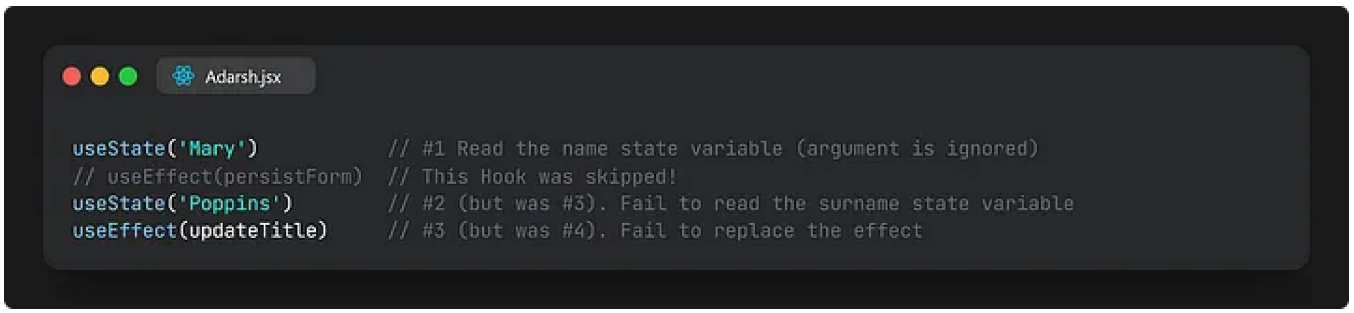
This means you are ensuring that Hooks are called in the same order each time a component renders. This is helpful in preserving your app’s state between multiple `useState` and `useEffect` calls.

So you should always make sure to use Hooks at the top level of your function before any early returns. React relies on the order in which Hooks are called, so if we have Hook calls in the same order every time, it looks like this:



But if we put a Hook call of `persistForm` inside a condition:

Then we are breaking the first rule by using a Hook inside a condition! Now the order changes as follows:



React doesn't know what to return for the second `useState` call. Hence multiple Hooks were skipped causing issues.

2. Only Call Hooks from React Functions: this rule states that:

- Don’t call Hooks from regular JavaScript functions.

By following this rule, we ensure that all stateful logic code of a component is clearly visible from its source code.

Instead of calling them from regular functions, we can:

- Call Hooks from React function components.
- Call Hooks from custom Hooks.

Intermediate level — React Hooks Cheat Sheet

1. Building custom Hooks

Apart from using the provided `useState` and `useEffect` Hooks, we can make a custom Hook as per the need to extract component logic into reusable functions.

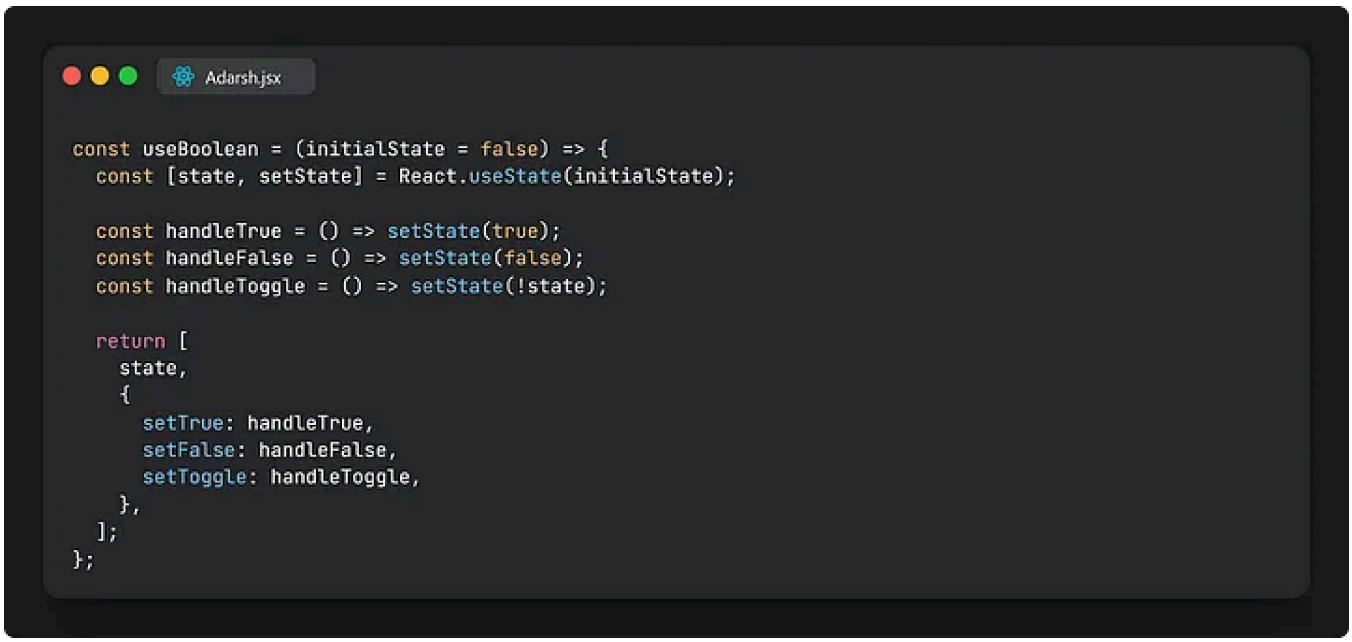
> Custom Hooks are some functions whose name starts with “use,” which may call other Hooks.

It doesn’t need to have a specific signature. The following points should be kept in mind when making a custom Hook:

1. Extracting a custom Hook: custom Hooks are used when we want to share logic between two JavaScript functions and then extract it to a third function. We should make sure to only call other Hooks unconditionally at the top level of the custom Hook.
   Here’s an example:

In this example, we see that our `useFriendStatus` Hook uses the `useState` and `useEffect` Hook to function along with some custom code to subscribe to a friend's status.

2. Using a custom Hook: after making a custom Hook, we can use it anywhere inside our components. For example, if we have the following custom Hook called `useBoolean`, which returns the state and the functions to update the state in an array:



Note that we are passing the `initialState` as an argument along with its default value (`false`). Back in the `App` component, we can make use of this Hook by passing

an initial state to it and by using its returned values to display the state and update it, as shown below:

2. The `useContext` Hook

The `useContext` Hook saves you from the stress of having to rely on a Context consumer.

> It accepts a context object and returns the current context value for that context.

The context object comes from `React.createContext`, and it's determined by the `value` prop of the nearest `<MyContext.Provider>` above the calling component in the tree.

Here's its call signature:

React's Context is initialized with the `createContext` top-level API. Here's an example:

Adarsh.jsx

```jsx
import React from 'react';

const CurrencyContext = React.createContext(null);

export { CurrencyContext };
```

The `createContext` function takes an initial value which is also the default value if no `value` prop is defined. We can use it in our example `Book` component as shown:

Adarsh.jsx

```jsx
const Book = ({ item }) => {
  return (
    <CurrencyContext.Consumer>
      {(currency) => (
        <li>
          {item.title} - {item.price} {currency}
        </li>
      )}
    </CurrencyContext.Consumer>
  );
};
```

The `useState` Hook takes the Context as a parameter to retrieve the `value` from it. Now, if we want to write the above `Book` component from `useContext` we end up with the following:

Adarsh.jsx

```jsx
const Book = ({ item }) => {
  const currency = React.useContext(CurrencyContext);

return (
    <li>
      {item.title} - {item.price} {currency}
    </li>
  );
};
```

3. The `useReducer` Hook

> The `useReducer` Hook is an alternative to `useState`. It is usually preferable when you have a complex state logic that involves multiple sub-values or when the state

> depends on the previous one.

It has the following call signature:

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

It accepts a reducer of type `(state, action) => newState` and returns the current state paired with a `dispatch` method. Here's an example where this Hook is used:

```jsx
const initialState = { width: 15 }
const reducer = (state, action) => {
  switch (action) {
    case 'plus':
      return { width: state.width + 15 }
    case 'minus':
      return { width: Math.max(state.width - 15, 2) }
    default:
      throw new Error("what's going on?" )
  }
}
const Bar = () => {
  // `useReducer` call
  const [state, dispatch] = useReducer(reducer, initialState)
  return <>
    <div style={{ background: 'teal', height: '30px', width: state.width }}></div>
    <div style={{marginTop: '3rem'}}>
        <button onClick={() => dispatch('plus')}>Increase bar size</button>
        <button onClick={() => dispatch('minus')}>Decrease bar size</button>
    </div>
    </>
}
render(Bar)
```

We can initialize the state lazily by passing an `init` function as its third argument. Whatever is returned from this function is returned as the state object. Here's an example to illustrate this:

Adarsh.jsx

```
// Adding a new function
const initializeState = () => ({
  width: 100
})
const initialState = { width: 15 }
const reducer = (state, action) => {
  switch (action) {
    case 'plus':
      return { width: state.width + 15 }
    case 'minus':
      return { width: Math.max(state.width - 15, 2) }
    default:
      throw new Error("what's going on?" )
  }
}
const Bar = () => {
  // Passing `initializeState` as third argument
  const [state, dispatch] = useReducer(reducer, initialState, initializeState)
  return <>
    <div style={{ background: 'teal', height: '30px', width: state.width }}></div>
    <div style={{marginTop: '3rem'}}>
        <button onClick={() => dispatch('plus')}>Increase bar size</button>
        <button onClick={() => dispatch('minus')}>Decrease bar size</button>
    </div>
    </>
}
render(Bar)
```

4. The `useCallback` Hook

> The `useCallback` Hook is used to return a memoized version of the passed callback that only changes if one of the dependencies has changed.

It’s useful when passing callbacks to optimized child components to prevent unnecessary renders. Here’s its call signature:

Adarsh.jsx

```
const memoizedCallback = useCallback(function, arrayDependency)
```

To use `useCallback` with an inline function, we can call it as follows:

Here, the `<Instructions />` child component has the `doSomething` prop, which is passed with a call to the `useCallback` Hook.

5. The `useMemo` Hook

> The `useMemo` Hook allows memoizing expensive functions to avoid calling them on every render.

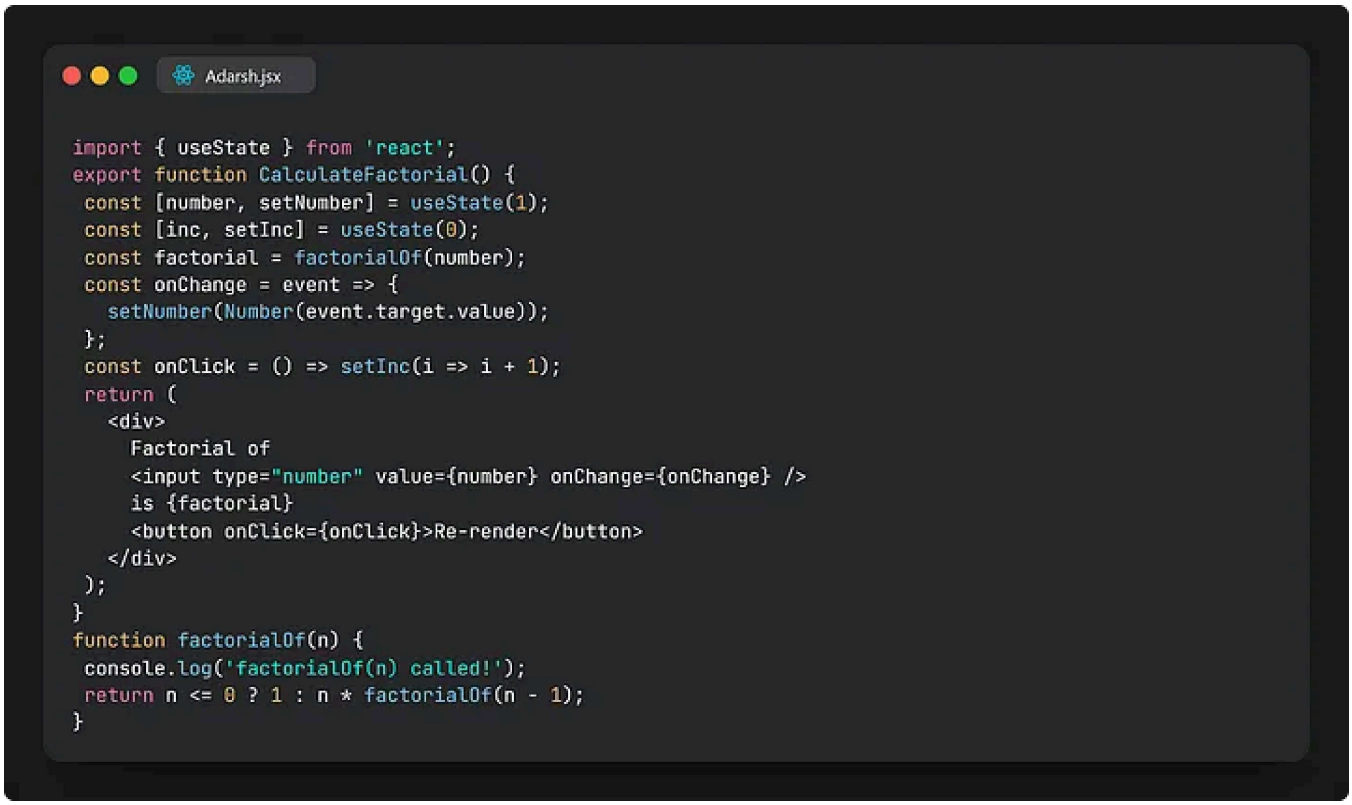
You pass a “create” function and an array of dependencies, and it will spit out its memoized value. This Hook has the following call signature:

The following points should be kept in mind when using the `useMemo` Hook:

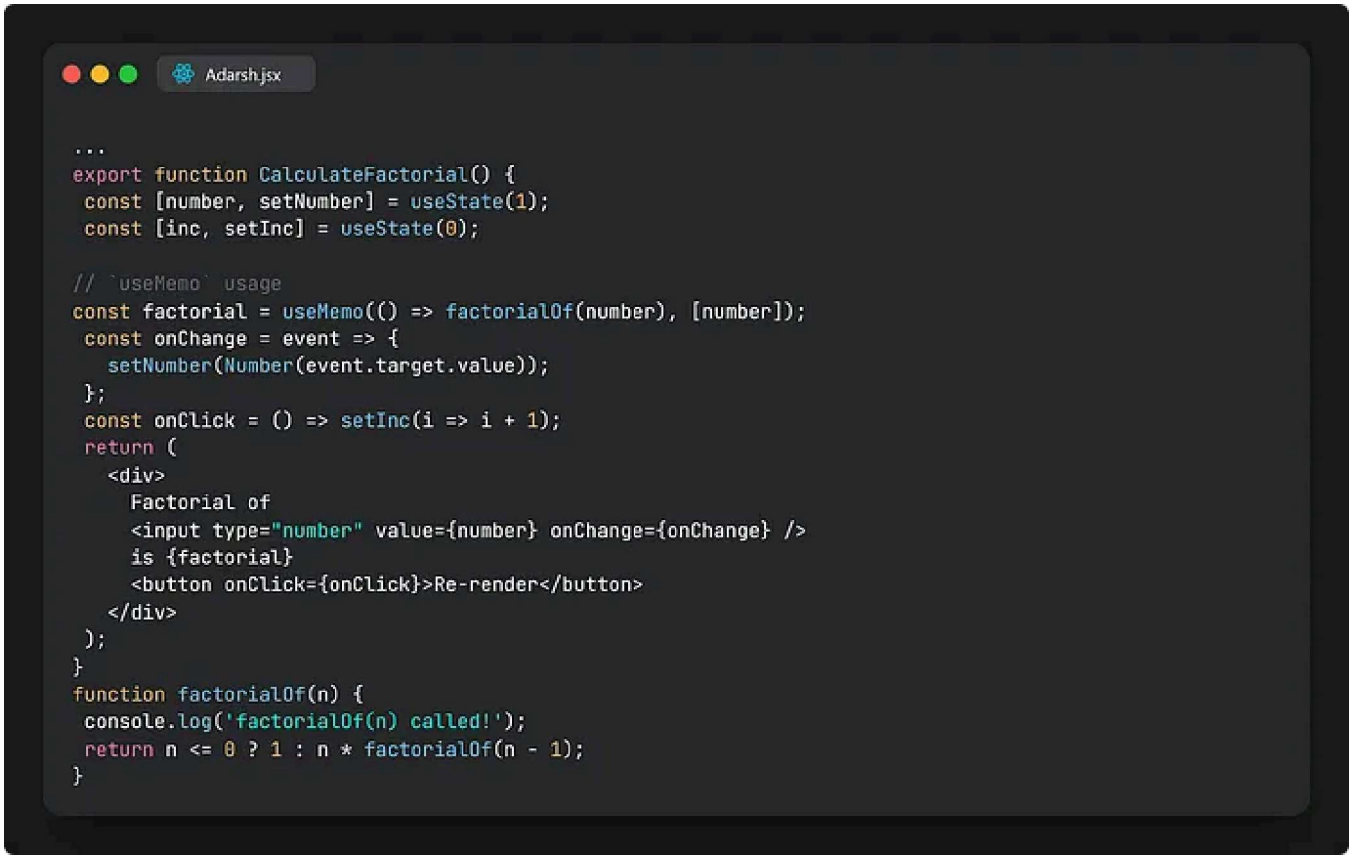
1. It will only recompute the memoized value when one of the dependencies you pass has changed.
2. We should not add side effects to its call; they should instead belong to the `useEffect` Hook.
3. During the initial rendering, `useMemo` invokes `compute`, memoizes the result, and returns to its component.

4. If we don't provide any array, a new value will be computed on every render of the component where `useMemo` is used.

Let's say we have a `<CalculateFactorial />` component:



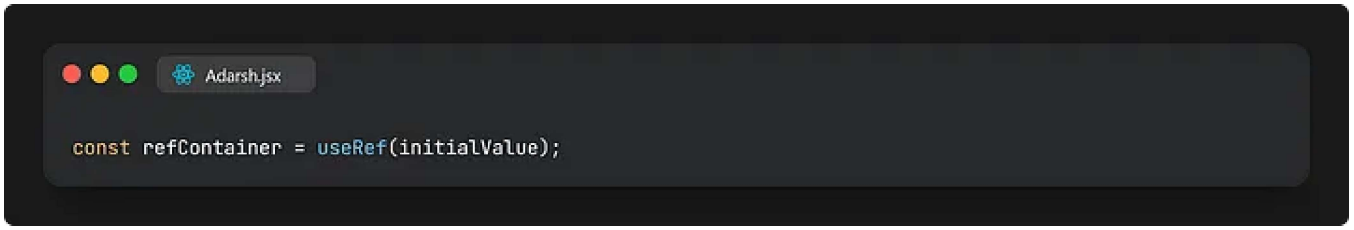
We can memoize the factorial calculation here whenever the component re-renders by using `useMemo(() => factorialOf(number), [number])` as shown in the updated code below:



6. The `useRef` Hook

> The `useRef` Hook returns a mutable `ref` object whose `.current` property is initialized to the passed argument `(initialValue)`.

It has the following call signature:



The following points are essential about `useRef`:

1. `useRef` acts as a "box" that can hold a mutable value in its `.current` property.
2. This Hook creates a plain JavaScript object. The difference between `useRef()` and creating a `{ current: ... }` object yourself is that `useRef` gives you the same `ref` object on *every render*.
3. If you mutate the `.current` property, it will not cause a re-render. The Hook doesn't notify you when its content changes.

4. The value of the reference is *persisted*, i.e., it stays the same between component re-renderings.

5. References can also access DOM elements. For that, we need to refer to the `ref` attribute of the element you would like to access.

Here's an example where we need to access the DOM element to focus on the input field when the component mounts:

After creating a reference to hold the input element in `inputRef` , it's assigned to the input field's `ref` attribute. After mounting, React sets `inputRef.current` to be the input element.

Advance level — React Hooks Cheat Sheet

1. Testing React Hooks

If your testing solution doesn't rely on React internals, then testing components with Hooks isn't different from how we usually test components.

Let's assume we have the following component, which counts the number of times you clicked the button:

We can now test this using React DOM or the React Testing Library to reduce the boilerplate code. But let's see how to test it with vanilla code. To make sure the behavior matches with what happens in the browser, we will wrap the code rendering and updating into multiple `ReactTestUtils.act()` calls as shown:

Adarsh.jsx

```
// counter.test.js

import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // Test first render and effect
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });

  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('You clicked 0 times');
  expect(document.title).toBe('You clicked 0 times');

  // Test second render and effect
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });

  expect(label.textContent).toBe('You clicked 1 times');
  expect(document.title).toBe('You clicked 1 times');
});
```

2. Data fetching with React Hooks

Data fetching is quite a typical pattern we see when working with a React app. But if you are using Hooks in a particular app component, then here's how you should fetch the data.

First, let's say we have this basic `App` component that shows a list of items:

Adarsh.jsx

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

function App() {
  const [data, setData] = useState({ hits: [] });

  return (
    <ul>
      {data.hits.map(item => (
        <li key={item.objectID}>
          <a target="_blank" href={item.url}>{item.title}</a>
        </li>
      ))}
    </ul>
  );
}

export default App;
```

Both the state and the state update function come from the `useState` Hook. Now, let's use the Axios library to fetch data here. We can implement our effect Hook as follows:

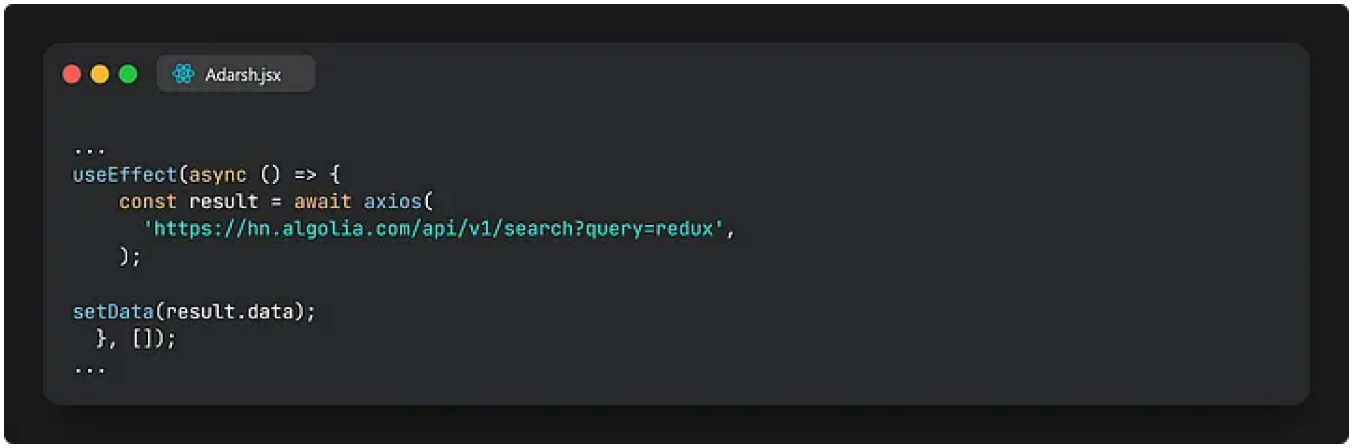
Adarsh.jsx

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

function App() {
  const [data, setData] = useState({ hits: [] });

  useEffect(async () => {
    // Using Axios to fetch data from an API endpoint
    const result = await axios(
      'https://hn.algolia.com/api/v1/search?query=redux',
    );

    setData(result.data);
  });

  return (
    <ul>
      {data.hits.map(item => (
        <li key={item.objectID}>
          <a target="_blank" href={item.url}>{item.title}</a>
        </li>
      ))}
    </ul>
  );
}

export default App;
```

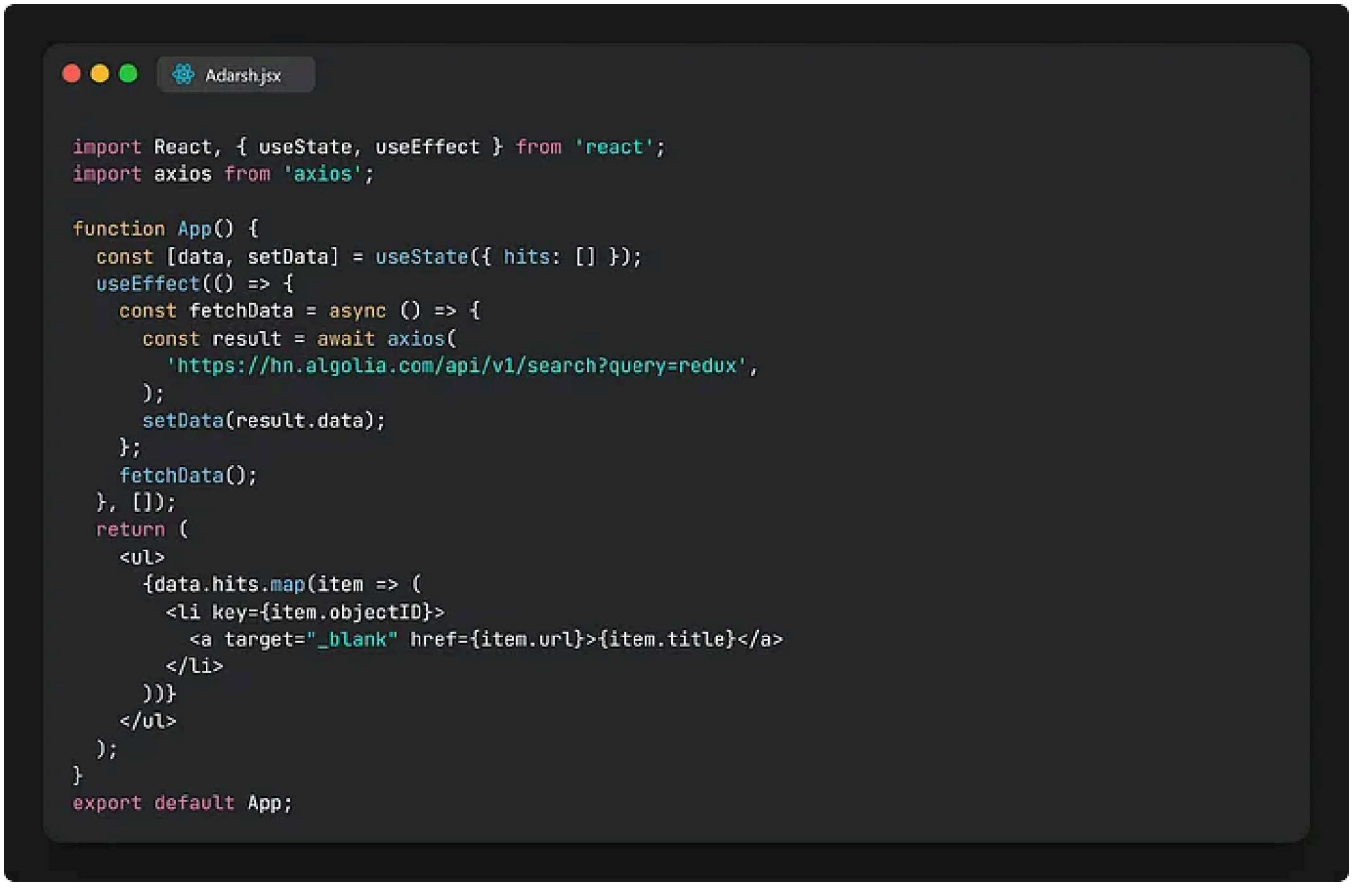
But if you run the app, you will see the code is stuck in a loop. The effect Hook runs when the component mounts but also when it updates. We are setting the state after

every data fetch. To fix this, we only want to fetch the data when the component mounts.

Hence, we should provide an empty array as the second argument to the `useEffect` Hook as shown:



To further improve the code quality, we can use an `async` function inside `useEffect` so that it returns a cleanup function in the Promise. So our final code will be:

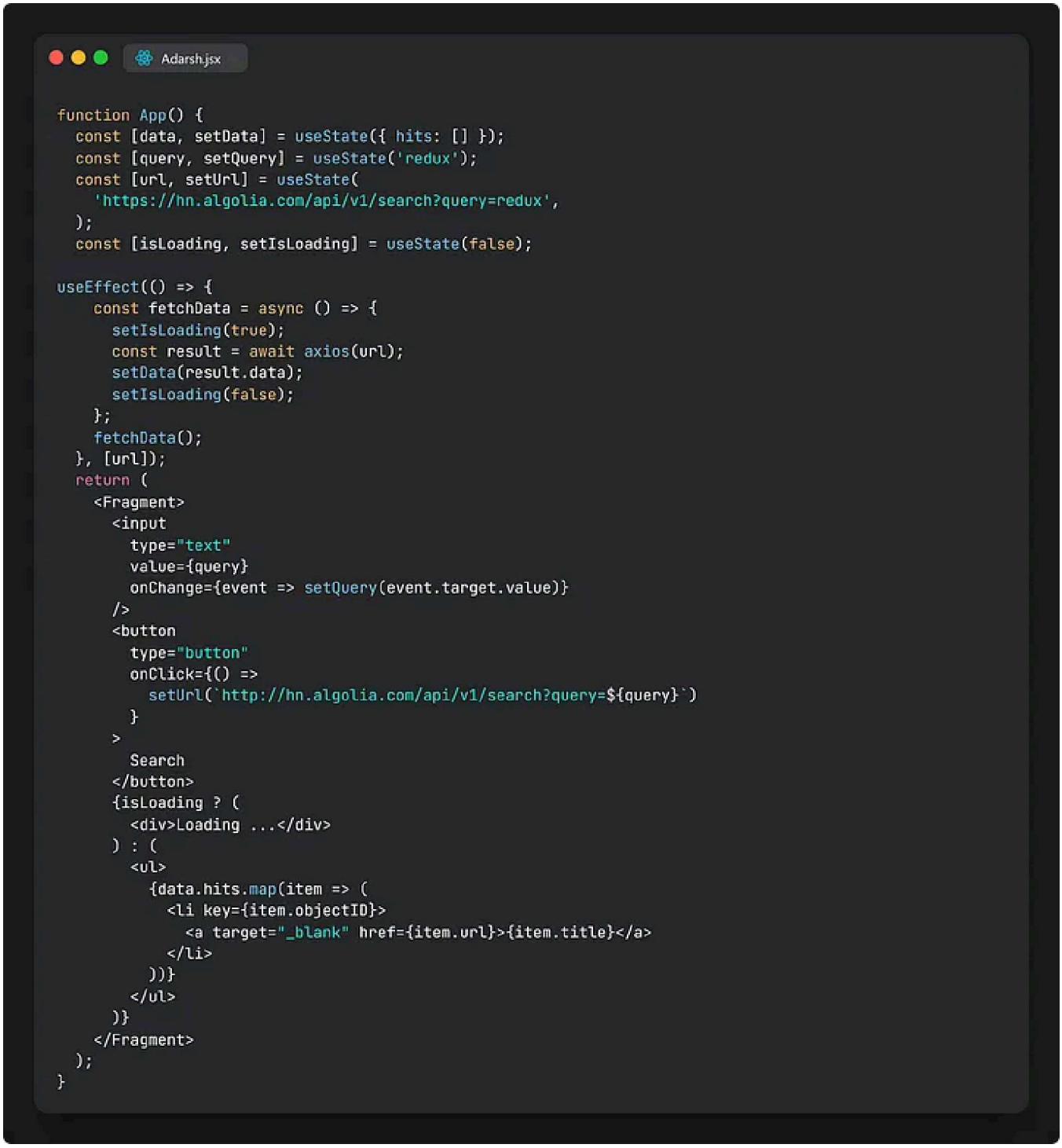


3. Creating a loading indicator with Hooks

Using the same data loading example as above, we can start implementing a basic loading indicator. For this, we will add another state value using the `useState` Hook:

Now inside of our `useEffect` call, we can set the toggle for the `true` and `false` values for `setIsLoading`:

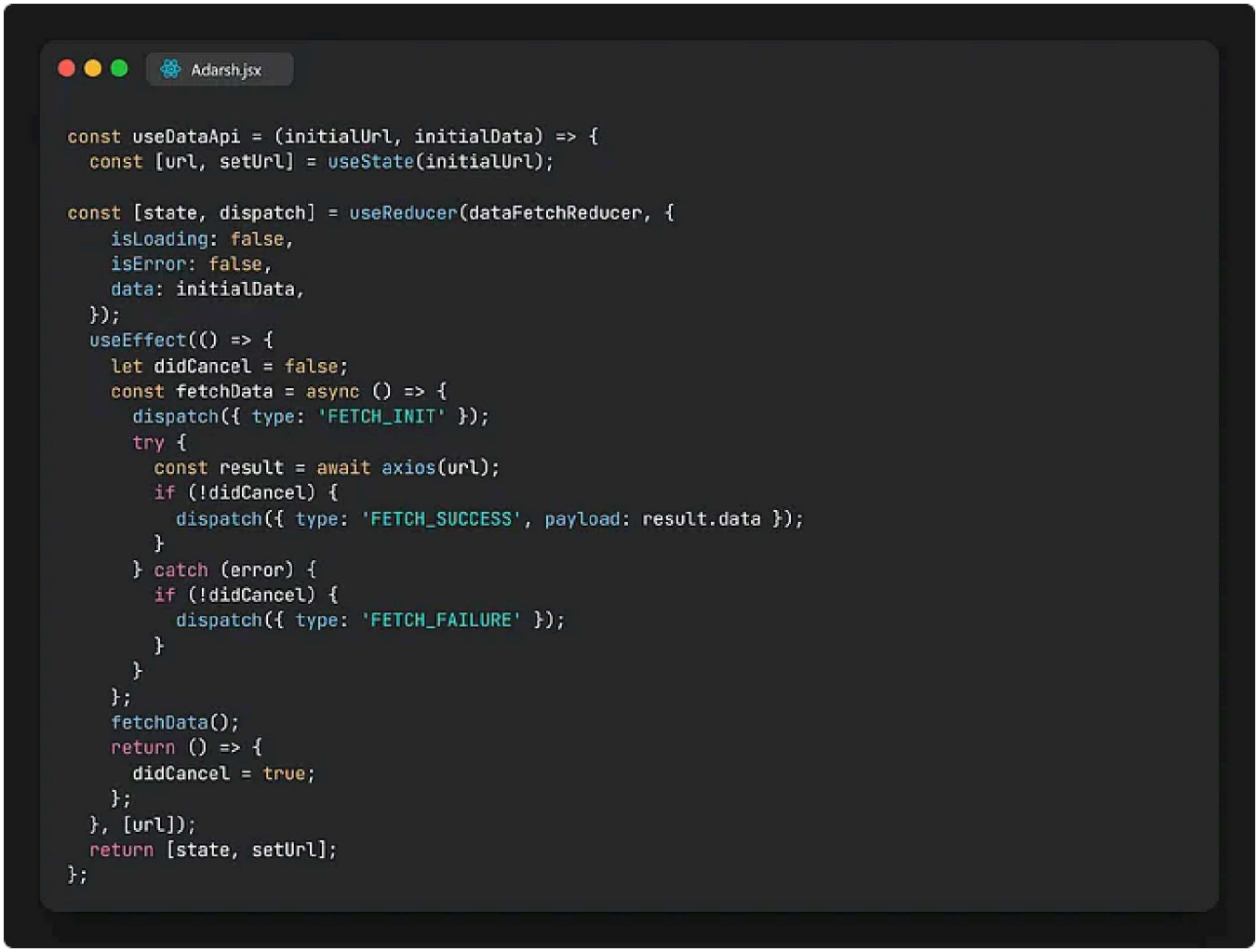
Once the effect is called for data fetching when the component mounts or the URL state changes, the loading state is set to true. Here's the entire code after these changes:



4. Data aborting using Hooks

We often see that the component state is set even though the component was already mounted. Due to this, we also want to abort data fetching in our component so that we don't encounter any bugs or loops.

We can use the effect Hook, which comes with a cleanup function and runs when a component unmounts. Let's use it to prevent data fetching in the following example:



Here, we used a boolean called `didCancel` so that the data fetching code knows about the component's state (mounted or unmounted). If it unmounts, the flag is set to `true`, which prevents setting the component state after the data fetching has been resolved asynchronously.

5. Getting previous props or states with Hooks

Sometimes we need previous props of a component to clean up a used effect. To illustrate this, let's say we have an effect that subscribes to a socket based on the `userId` props. If this prop changes, we want to unsubscribe from the *previously* set `userId` and then subscribe to the *next* one.

For this, we can use a cleanup function with the `useEffect` function as:

Here, first the `ChatAPI.unsubscribeFromSocket(3)` will run and then `ChatAPI.unsubscribeFromSocket(4)` in order when `userId` changes. Hence, the cleanup function does the work for us. It captures the previous `userId` in a closure.

Conclusion — React Hooks Cheat Sheet

In this React Hooks cheat sheet, we covered all the major topics related to Hooks. From why it is needed, to how to unlock Hook's superpowers to manage and handle your app state effectively, everything is covered with relevant examples.

We hope this cheat sheet will help you in your future or previous React applications.

React | Reactjs | React Hook | Web Development | Website

More from The Expert Developer

The Expert Developer

🚀 The Complete React.js 🔭Developer Roadmap for 2025 🧑‍💻

💻 “React is not just a library; it’s an ecosystem!”—🤖If you want to become a top-tier 🔋 React.js developer in 2025, you need to...

Feb 20

The Expert Developer

🚀 The Complete Flutter Developer Roadmap in 2025 🧑‍💻

📱Flutter is the future of cross-platform development. But where do you start? 🤖Here's your complete roadmap to becoming a Flutter...

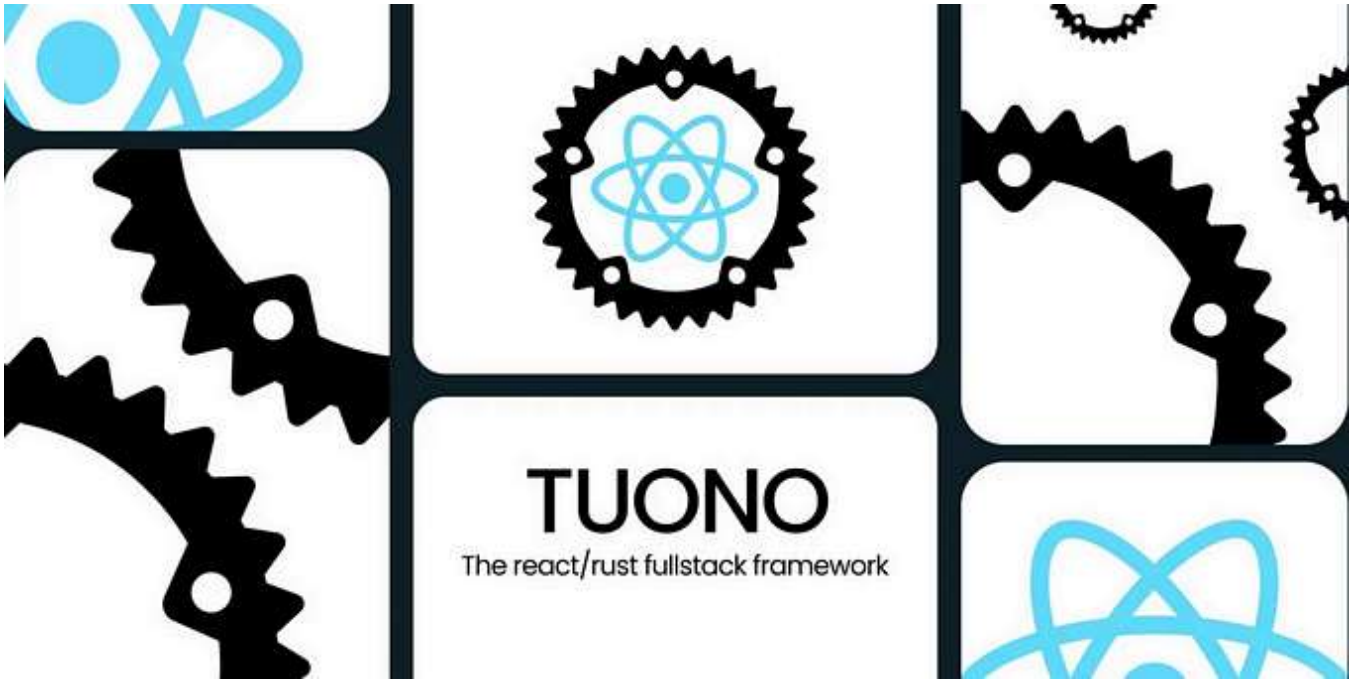
Feb 20

The Expert Developer

🚀 10 Common JavaScript 🔭Mistakes (And How to Avoid Them!) 📠

🎻 JavaScript is incredibly powerful and flexible. However, this same flexibility often leads to common pitfalls. Understanding and...

Mar 8 2



The Expert Developer

🎩 Introducing Tuono: 🌓 Next.js-Style Framework for Rust Backend 🚀🦀

🔝 It's an exciting development in web frameworks, especially for developers comfortable with Next.js 🐆 but want to take advantage 🎃 of the...

Feb 19 👏 37

See all from The Expert Developer

Recommended from Medium

In Dev Simplified by Neha Gupta

Why Companies Are Saying GoodBye to Next.js?

Are you using Next.js or planning to for your next project? Then you need to know this before making a decision!

3d ago 364 23

Developersanchit

Frontend Coding Interview : JavaScript Challenges Companies Frequently Ask

When preparing for your next coding interview, it's important to understand core JavaScript concepts and be able to implement practical...

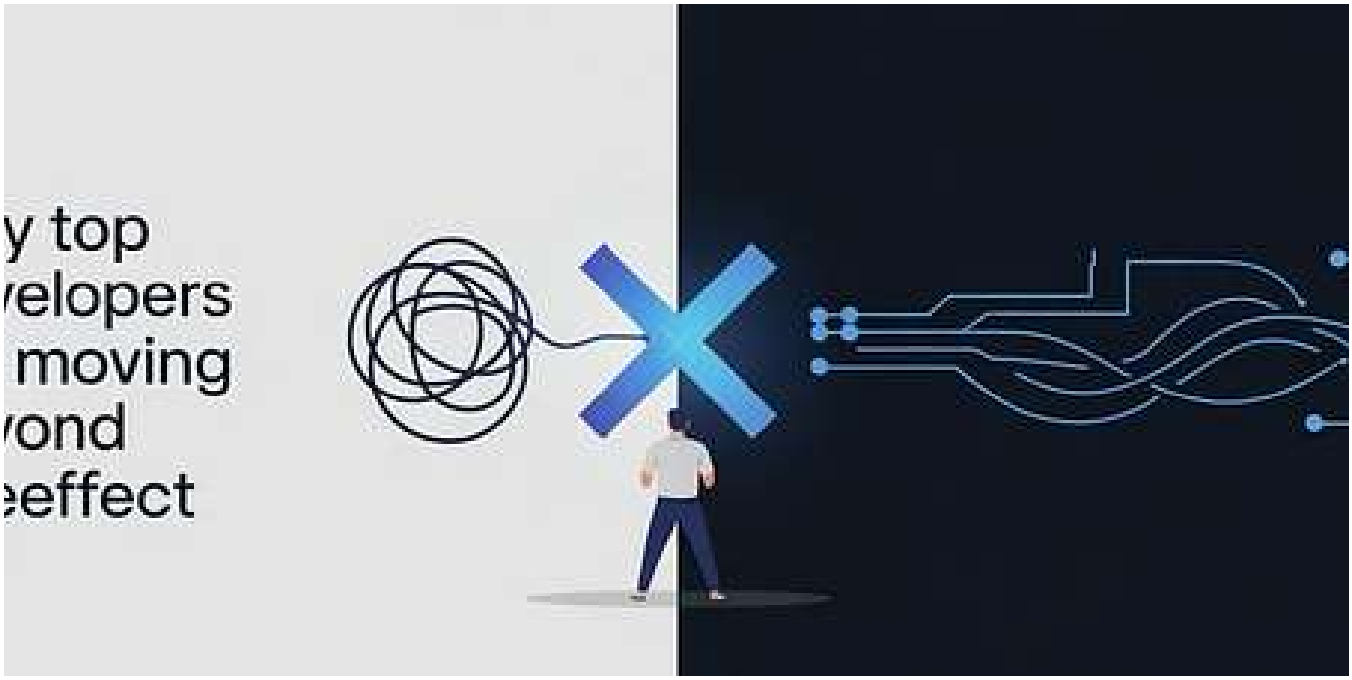
Mar 29 93 2

Shahmir Khan

React Patterns Every Developer Should Know

React is a powerful library for building user interfaces, and over time, several patterns have emerged to help developers write cleaner...

Mar 27 44



In The Syntax Diaries by Amaresh Adak

Stop Using useEffect Like This—Here's What React Architects Do Instead

Why Top Developers Are Moving Beyond useEffect (And You Should Too)

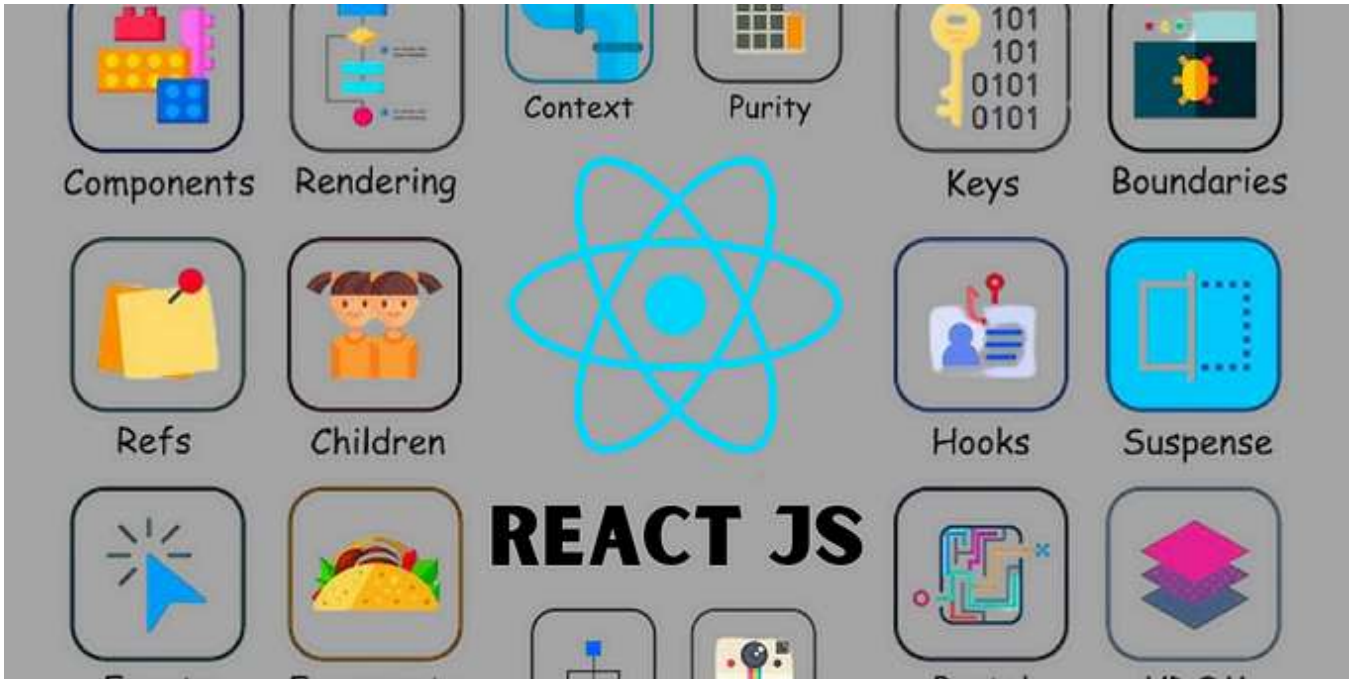
6d ago 133 5

Vitaliy Korzhenko

The Most Difficult JavaScript Interview Question

Can You Answer It ?

Dec 12, 2024 267 8



In Full Stack Forge by Daniel Scott

All React Concepts Explained in Just One Read

I'm sure there's something here you didn't know—get my 8 years of experience!

Mar 16