Mastering Side Effects In React

Next

Side effects can include fetching data from an API, modifying the DOM, subscribing to events, or updating the local storage. Identifying and handling side effects properly is key to ensuring your app behaves predictably.

FredolyWagni

Conditional Side Effects

Sometimes, you may want to perform a side effect only under specific conditions. Use conditional statements within the useEffect hook to control when the side effect should be executed.

```jsx
import { useEffect } from 'react';

function ExampleComponent({ condition }) {
  useEffect(() => {
    if (condition) {
      // Side effect code here
      // e.g., perform an action if condition is true
    }
  }, [condition]);

  return <div>Your component</div>;
}
```

Next

Clean Up Side Effects

It's essential to clean up any side effects when a component is unmounted or when specific dependencies change. This prevents memory leaks and avoids potential issues. You can return a cleanup function within the useEffect hook to handle this.

```jsx
import { useEffect } from 'react';

function ExampleComponent() {
  useEffect(() => {
    // Side effect code here
    // e.g., event listeners or subscriptions

    return () => {
      // Cleanup code here
      // e.g., remove event listeners or unsubscribe
    };
  }, []);

  return <div>Your component</div>;
}
```

Next

FredolyWagni

Dependency Array

By specifying dependencies in the useEffect hook, you can control when the side effect should run. Only update the side effect when specific dependencies change, avoiding unnecessary re-renders.

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

function ExampleComponent({ userId }) {
  const [userData, setUserData] = useState({});

  useEffect(() => {
    // Side effect code here
    // e.g., fetching user data based on ID
    fetchUserData(userId).then((result) => setUserData(result));
  }, [userId]); // Update only when userId changes

  return <div>Your component</div>;
}
```

Next

FredolyWagni

Debouncing and Throttling

When dealing with side effects triggered by user events like keystrokes or scroll events, consider debouncing or throttling to optimize performance. Debouncing delays the execution of a function, while throttling limits the number of times it can run within a specified timeframe.

```jsx
import { useEffect } from 'react';
import { debounce, throttle } from 'lodash';

function ExampleComponent() {
  useEffect(() => {
    const handleScroll = debounce(() => {
      // Side effect code here
      // e.g., updating UI based on scroll position
    }, 250); // Debounce time in milliseconds

    window.addEventListener('scroll', handleScroll);

    return () => {
      window.removeEventListener('scroll', handleScroll);
    };
  }, []);

  return <div>Your component</div>;
}
```

Next

FredolyWagni

Custom Hooks

If you find yourself reusing the same side effect logic across multiple components, consider creating custom hooks. Custom hooks encapsulate the side effect logic, promoting code reuse and keeping your components clean and concise.

```jsx
import { useEffect } from 'react';

function useCustomSideEffect(dependency) {
  useEffect(() => {
    // Side effect code here
    // e.g., perform common side effect logic
  }, [dependency]);
}

function ExampleComponent({ data }) {
  useCustomSideEffect(data);

  return <div>Your component</div>;
}
```