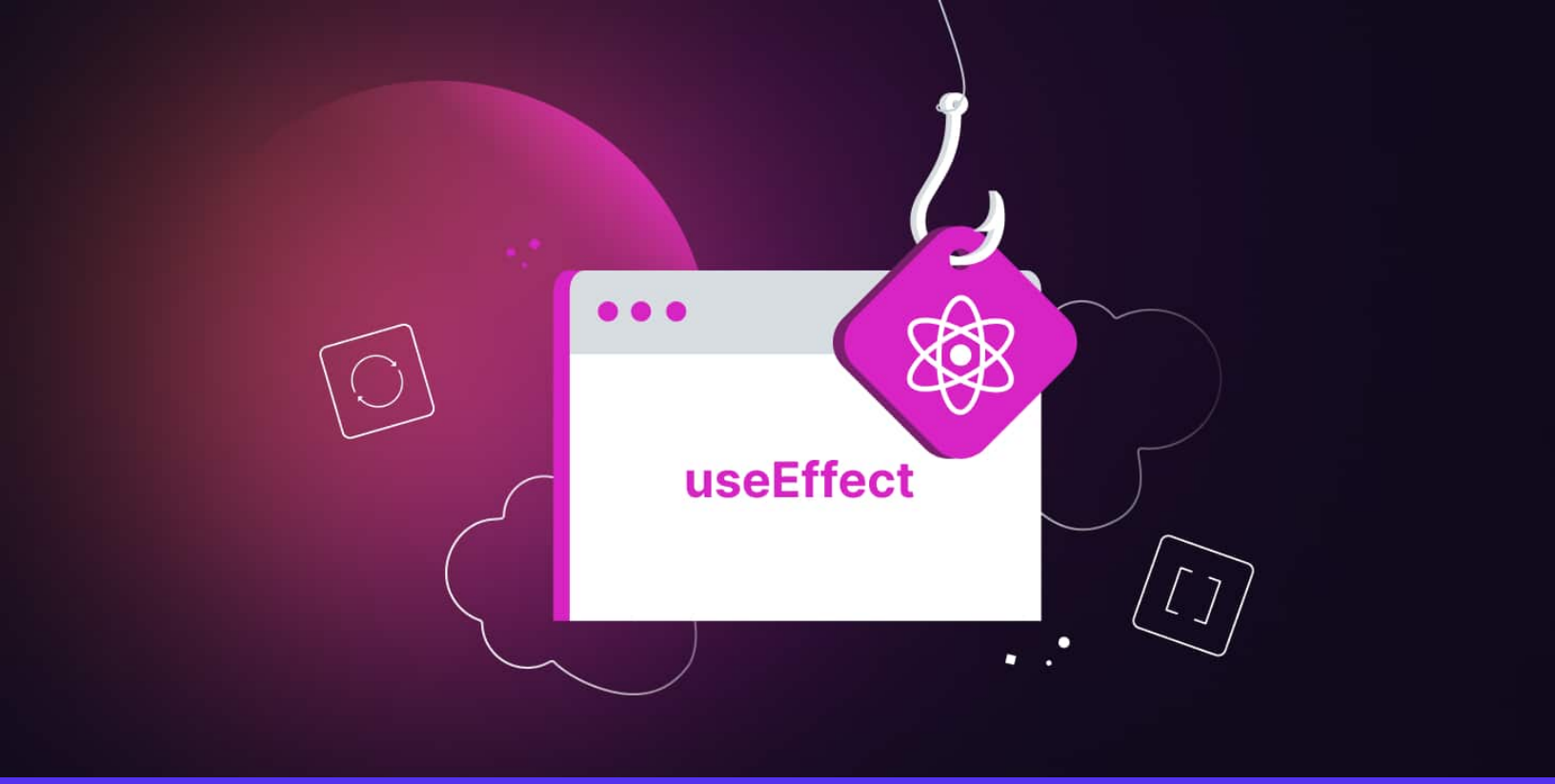

Demystifying React’s useEffect Hook

React is one of the popular JavaScript libraries for building user interfaces. When building these interfaces, you may need to perform side effects, such as fetching data from an API, subscribing to events, or manipulating the DOM.

That's where the powerful `useEffect` Hook comes into play. It allows you to seamlessly handle these side effects declaratively and efficiently, ensuring your UI stays responsive and up-to-date.

Whether you're new to React or an experienced developer, understanding and mastering `useEffect` is essential for creating robust and dynamic applications. In this article, you'll learn how the `useEffect` Hook works and how to use it in your React project.

What Is Side-Effect in React?

When working with React components, there are times when we need to interact with entities or perform actions outside of React's scope. These external interactions are known as **side effects**.

In React, most components are pure functions, meaning they receive inputs (props) and produce predictable output (JSX), as seen in the example below:

```
export default function App() {
  return <User userName="JaneDoe" />
}

function User(props) {
  return <h1>{props.userName}</h1>; // John Doe
}
```

However, side effects are not predictable because they involve interactions outside of React's usual scope.

Consider an example where you want to dynamically change the title of the browser tab to display the user's `userName`. While it might be tempting to do this directly within the component, it's not the recommended approach because this is considered a side effect:

```
const User = ({ userName }) => {
  document.title = `Hello ${userName}`; // ?Never do this in the componen

  return <h1>{userName}</h1>;
}
```

Performing side effects directly within the component body can interfere with the rendering process of our React component.

To avoid interference, you should separate side effects so they only render or function after our component has rendered, ensuring a clear separation between the rendering process and any necessary external interactions. This separation is done with the `useEffect` Hook.

Understanding the Basics of useEffect

The `useEffect` Hook is designed to mimic lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` found in class components.

To use `useEffect`, you will need to import it from “react” and then call it within a function component (at the top level of the component). It takes two arguments: a callback function and an optional dependency array.

```
useEffect(callbackFn, [dependencies]);
```

This can be better written as:

```
useEffect(() => {
  // code to run when the effect is triggered
}, [dependencies]);
```

- The **callback function** contains the code to run when the component renders or the dependency value changes. This is where you perform side effect(s).
- The **dependency array** specifies the values that should be monitored for changes. The callback function will be executed when any value in this array changes.

For example, you can now correct the previous example to perform the side effect properly within a `useEffect` Hook:

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

const User = ({ userName }) => {
  useEffect(() => {
    document.title = `Hello ${userName}`;
  }, [userName]);

  return <h1>{userName}</h1>;
}
```

In the example above, the `useEffect` Hook will be called after the component has rendered and whenever the dependency — `userName`'s value — changes.

Working With Dependencies In useEffect

Dependencies play a crucial role in controlling the execution of `useEffect`. It is the second argument of the `useEffect` Hook.

```
useEffect(() => {
  // code to run when the effect is triggered
}, [dependencies]);
```

Using an empty dependency array `[]` ensures the effect runs only once, simulating `componentDidMount`. Specifying dependencies correctly allows the effect to update when specific values change, similar to `componentDidUpdate`.

Note: You should take care when dealing with complex dependencies. Unnecessary updates can be avoided by carefully selecting which values to include in the dependency array.

Omitting the dependency array altogether will cause the effect to run every time the component renders, which can lead to performance issues.

```
useEffect(() => {
  // code to run when the effect is triggered
});
```

In React, understanding how rendering works is a huge plus because you will be able to know the importance of the dependency array.

How Does Rendering Work in React?

In React, rendering generates the user interface (UI) based on a component's current state and props. There are different scenarios where rendering occurs. The initial render happens when a component is first rendered or mounted.

Aside from this, a change in the `state` or `props` of a component triggers a re-render to ensure that the UI reflects the updated values. React applications are built with a tree-like structure of components, forming a hierarchy. React starts from the root component during rendering and recursively renders its child components.

This means all components would be rendered if a change occurs in the root component. It's important to note that calling side effects(which are most times expensive functions) on every render can be costly. To optimize performance, you can use the dependency array in the `useEffect` Hook to specify when it should be triggered, limiting unnecessary re-renders.

Advanced Usage of useEffect: Cleaning Up Side Effects

The `useEffect` Hook allows us to perform side effects and provides a mechanism to clean up those side effects. This ensures that any resources or subscriptions created during the side effect are properly released and prevents memory leaks.

Let's explore how you can clean up side effects using the `useEffect` Hook:

```
useEffect(() => {
  // Perform some side effect

  // Cleanup side effect
  return () => {
    // Cleanup tasks
  };
}, []);
```

In the code snippet above, the cleanup function is defined as a return value within the `useEffect` Hook. This function is invoked when the component is about to unmount or before a subsequent re-render occurs. It allows you to clean up any resources or subscriptions established during the side effect.

Here are some examples of advanced usage of the `useEffect` Hook for cleaning up side effects:

1. Clearing Intervals

```
useEffect(() => {
    const interval = setInterval(() => {
        // Perform some repeated action
    }, 1000);
    return () => {
        clearInterval(interval); // Clean up the interval
    };
}, []);
```

In this example, we set up an interval that performs an action every second. The cleanup function clears the interval to prevent it from running after the component is unmounted.

2. Cleaning Event Listeners

```
useEffect(() => {
    const handleClick = () => {
        // Handle the click event
    };

    window.addEventListener('click', handleClick);

    return () => {
        window.removeEventListener('click', handleClick); // Clean up the
    };
```

```
  }, []);
```

Here, we create an event listener for the click event on the window object. The cleanup function removes the event listener to avoid memory leaks and ensure proper cleanup.

Remember, the cleanup function is optional, but it is highly recommended to clean up any resources or subscriptions to maintain a healthy and efficient application.

Using the useEffect Hook

The `useEffect` Hook enables you to perform tasks that involve interacting with external entities or APIs, such as web APIs like localStorage or external data sources.

Let's explore the usage of the `useEffect` Hook with various scenarios:

1. Working with Web APIs (localStorage)

```
useEffect(() => {
 // Storing data in localStorage
  localStorage.setItem('key', 'value');
  // Retrieving data from localStorage
  const data = localStorage.getItem('key');
  // Cleanup: Clearing localStorage when component unmount
  return () => {
    localStorage.removeItem('key');
  };
```

```
  }, []);
```

In this example, the `useEffect` Hook is used to store and retrieve data from the browser's localStorage. The cleanup function ensures that the localStorage is cleared when the component is unmounted (this may not be a good use case always as you may want to keep the localStorage data until the browser is refreshed).

2. Fetching Data From an External API

```
useEffect(() => {
  // Fetching data from an external API
  fetch('https://api.example.com/data')
    .then((response) => response.json())
    .then((data) => {
      // Do something with the data
    });
}, []);
```

Here, the `useEffect` Hook is used to fetch data from an external API. The fetched data can then be processed and used within the component. It is not compulsory to add a cleanup function always.

Other Popular Side Effects

The `useEffect` Hook can be used for various other side effects, such as:

A. Subscribing to Events:

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

B. Modifying the Document Title:

```
useEffect(() => {
  document.title = 'New Title';
  return () => {
    document.title = 'Previous Title';
  };
}, []);
```

C. Managing Timers:

```
useEffect(() => {
  const timer = setInterval(() => {
    // Do something repeatedly
  }, 1000);
  return () => {
```

```
      clearInterval(timer);
    };
  }, []);
```

Common useEffect Errors and How To Avoid Them

While working with the `useEffect` Hook in React, it's possible to encounter errors that can lead to unexpected behavior or performance issues.

Understanding these errors and knowing how to avoid them can help ensure smooth and error-free usage of `useEffect`.

Let's explore some common `useEffect` errors and their solutions:

1. Missing Dependency Array

One common mistake is forgetting to include a dependency array as the second argument of the `useEffect` Hook.

ESLint, will always flag this as a warning because it can result in unintended behaviours, such as excessive re-rendering or stale data.

```
useEffect(() => {
  // Side effect code
}); // Missing dependency array
```

Solution: Always provide a dependency array to `useEffect`, even if it's empty. Include all the variables or values that the effect depends on. This helps React determine when the effect should run or be skipped.

```
useEffect(() => {
  // Side effect code
}, []); // Empty dependency array or with appropriate dependencies
```

2. Incorrect Dependency Array

Providing an incorrect dependency array can lead to issues as well. If the dependency array is not accurately defined, the effect might not run when the expected dependencies change.

```
const count = 5;
const counter = 0;
useEffect(() => {
  // Side effect code that depends on 'count'
  let answer = count + 15;
}, [count]); // Incorrect dependency array
```

Solution: Make sure to include all the necessary dependencies in the dependency array. If the effect depends on multiple variables, include all of them to trigger the effect when any of the dependencies change.

```
const count = 5;
useEffect(() => {
  // Side effect code that depends on 'count'
  let answer = count + 15;
}, [count]); // Correct dependency array
```

3. Infinite Loops

Creating an infinite loop can happen when the effect modifies a state or prop that is also dependent on the effect itself. This leads to the effect being triggered repeatedly, causing excessive re-rendering and potentially freezing the application.

```
const [count, setCount] = useState(0);
useEffect(() => {
  setCount(count + 1); // Modifying the dependency 'count' inside the eff
}, [count]); // Dependency array includes 'count'
```

Solution: Ensure that the effect does not directly modify a dependency that is included in its dependency array. Instead, create separate variables or use other state management techniques to handle necessary changes.

```
const [count, setCount] = useState(0);
useEffect(() => {
  setCount((prevCount) => prevCount + 1); // Modifying the 'count' using a
}, []); // You can safely remove the 'count' dependency
```

4. Forgetting Cleanup

Neglecting to clean up side effects can lead to memory leaks or unnecessary resource consumption. Not cleaning up event listeners, intervals, or subscriptions can result in unexpected behavior, especially when the component unmounts.

```
useEffect(() => {
  const timer = setInterval(() => {
    // Perform some action repeatedly
  }, 1000);
  // Missing cleanup
  return () => {
    clearInterval(timer); // Cleanup missing in the return statement
  };
}, []);
```

Solution: Always provide a cleanup function in the return statement of the `useEffect` Hook.

```
useEffect(() => {
  const timer = setInterval(() => {
    // Perform some action repeatedly
  }, 1000);
  return () => {
    clearInterval(timer); // Cleanup included in the return statement
  };
}, []);
```

By being aware of these common `useEffect` errors and following the recommended solutions, you can avoid potential pitfalls and ensure the correct and efficient usage of the `useEffect` Hook in your React applications.

Summary

React's `useEffect` Hook is a powerful tool for managing side effects in function components. Now that you have a deeper understanding of `useEffect`, it's time to apply

your knowledge and bring your React applications to life.

You can also make your React application run live by deploying to Kinsta's Application hosting for free!

Now it's your turn. What is your thought on the `useEffect` Hook? Please feel free to share it with us in the comments section below.