What is the useEffect hook?

You have been introduced to the primary usage of the `useEffect` hook, a built-in React hook best suited to perform side effects in your React components.

In this reading you will be introduced to the correct usage of the dependency array and the different `useEffect` calls that can be used to separate different concerns. You will also learn how you can clean up resources and free up memory in your `useEffect` logic by returning a function.

The code you place inside the `useEffect` hook always runs after your component mounts or, in other words, after React has updated the DOM.

In addition, depending on your configuration via the dependencies array, your effects can also run when certain state variables or props change.

By default, if no second argument is provided to the `useEffect` function, the effect will run after every render.

```
useEffect(() => {
  document.title = 'Little Lemon';
});
```

However, that may cause performance issues, especially if your side effects are computationally intensive. A way to instruct React to skip applying an effect is passing an array as a second parameter to `useEffect`.

In the below example, the integer variable `version` is passed as the second parameter. That means that the effect will only be re-run if the `version` number changes between renders.

```
useEffect(() => {
  document.title = `Little Lemon, v${version}`;
}, [version]); // Only re-run the effect if version changes
```

If `version` is 2 and the component re-renders and version still equals 2, React will compare `[2]` from the previous render and `[2]` from the next render. Since all items inside the array are the same, React would skip running the effect.

Use multiple Effects to Separate Concerns

React doesn't limit you in the number of effects your component can have. In fact, it encourages you to group related logic together in the same effect and break up unrelated logic into different effects.

```
function MenuPage(props) {
  const [data, setData] = useState([]);

  useEffect(() => {
    document.title = 'Little Lemon';
  }, []);

  useEffect(() => {
    fetch(`https://littlelemon/menu/${id}`)
      .then(response => response.json())
      .then(json => setData(json));
```

```
  }, [props.id]);

  // ...
}
```

Multiple hooks allow you to split the code based on what it is doing, improving code readability and modularity.

Effects with Cleanup

Some side effects may need to clean up resources or memory that is not required anymore, avoiding any memory leaks that could slow down your applications.

For example, you may want to set up a subscription to an external data source. In that scenario, it is vital to perform a cleanup after the effect finishes its execution.

How can you achieve that? In line with the previous point of splitting the code based on what it is doing, the `useEffect` hook has been designed to keep the code for adding and removing a subscription together, since it's tightly related.

If your effect returns a function, React will run it when it's time to clean up resources and free unused memory.

```
function LittleLemonChat(props) {
  const [status, chatStatus] = useState('offline');

  useEffect(() => {
    LemonChat.subscribeToMessages(props.chatId, () => setStatus('online'))

    return () => {
      setStatus('offline');
      LemonChat.unsubscribeFromMessages(props.chatId);
    };
  }, []);

  // ...
}
```

Returning a function is optional and it's the mechanism React provides in case you need to perform additional cleanup in your components.

React will make sure to run the cleanup logic when it's needed. The execution will always happen when the component unmounts. However, in effects that run after every render and not just once, React will also clean up the effect from the previous render before running the new effect next time.

Conclusion

In this lesson, you learned some practical tips for using the built-in Effect hook. In particular, you were presented with how to use the dependency array properly, how to separate different concerns in different effects, and finally how to clean up unused resources by returning an optional function inside the effect.