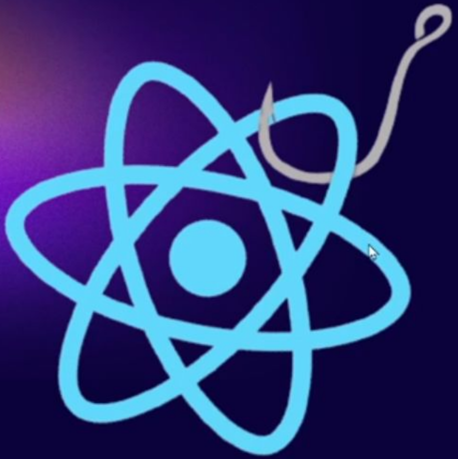
Deep dive into useEffect()



useEffect()

What are side effects in react?

- Not predictable
- Actions which are performed with the "outside world"
- A side effect is performed when we need to reach outside the scope of our current react components to do something
- React component rendering and side-effect logic are independent

Some common side effects -

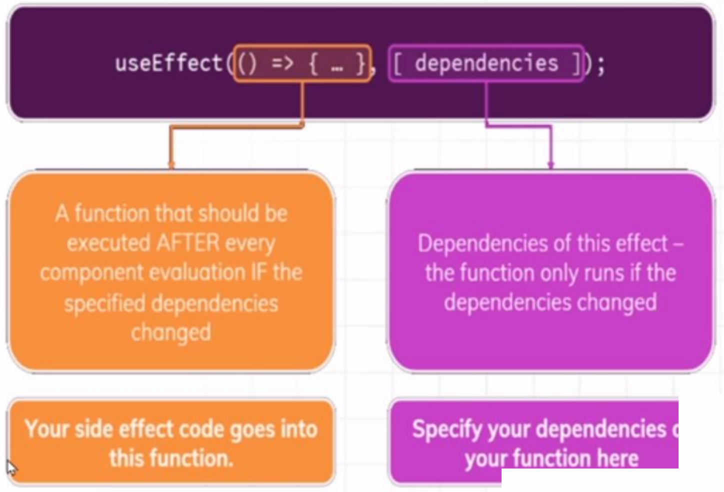
- Making a request to an API for data from a backend server
- To interact with browser APIs (that is, to use document or window directly) / Manipulating DOM directly
- Using unpredictable timing functions like setTimeout() setInterval()
- Reading data from local storage

What is useEffect?

useEffect exists –

- To provide a way to handle performing these side effects
- Doesn't affect the rendering or performance of the component that it's in
- Performs asynchronous tasks

useEffect Syntax



```
useEffect(() => { … }, [ dependencies ]);
```

A function that should be executed AFTER every component evaluation IF the specified dependencies changed

Dependencies of this effect – the function only runs if the dependencies changed

Your side effect code goes into this function.

**Specify your dependencies o
your function here**

What is the useEffect cleanup function?

- The useEffect cleanup allows us to tidy up our code before our component unmounts.

- When our code runs and reruns for every render, useEffect also cleans up after itself using the cleanup function.

- The cleanup function prevents **memory leaks** and **removes** some unnecessary and unwanted behaviors.

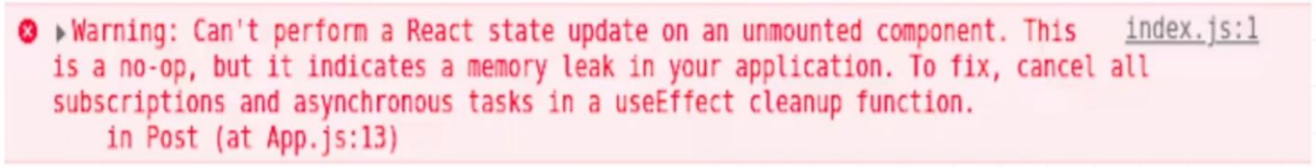
Why is the useEffect cleanup function useful?

- prevent unwanted behaviors and optimizes application performance.

- Let's look at this scenario: imagine we get a fetch of a particular user through a user's id, and, before the fetch completes, we change our mind and try to get another user. At this point, the prop, or in this case, the id, updates while the previous fetch request is still in progress.

When should we use the useEffect cleanup?

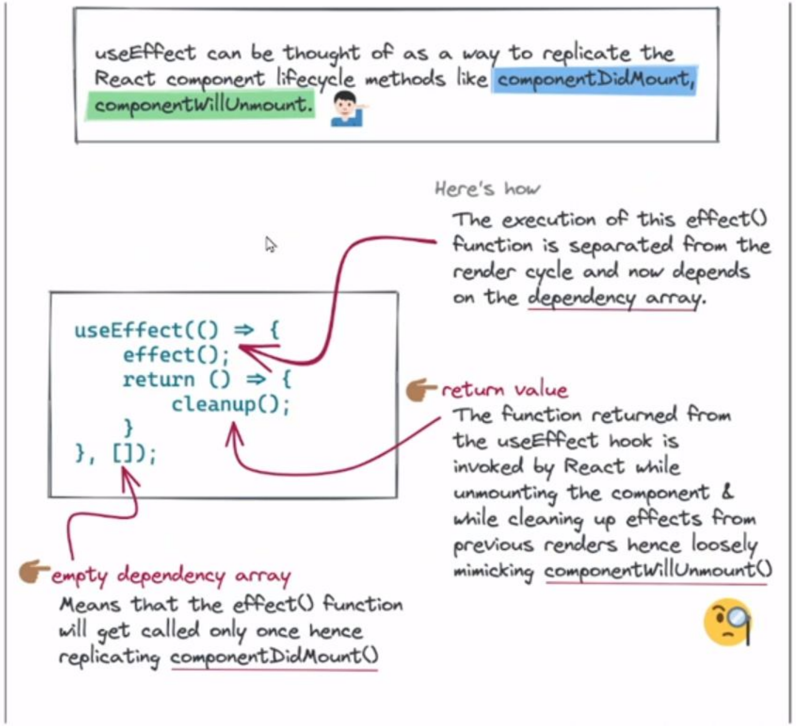
- If our component unmounts before our promise resolves, **useEffect** will try to update the state (on an unmounted component) and send an error that looks like this:



```
▸Warning: Can't perform a React state update on an unmounted component. This    index.js:1
is a no-op, but it indicates a memory leak in your application. To fix, cancel all
subscriptions and asynchronous tasks in a useEffect cleanup function.
    in Post (at App.js:13)
```

To fix this error, we use the cleanup function to resolve it.

useEffect Cleanup



useEffect can be thought of as a way to replicate the React component lifecycle methods like componentDidMount, componentWillUnmount. 🤷‍♂️

Here's how

The execution of this effect() function is separated from the render cycle and now depends on the dependency array.

```
useEffect(() => {
    effect();
    return () => {
        cleanup();
    }
}, []);
```

👉 return value

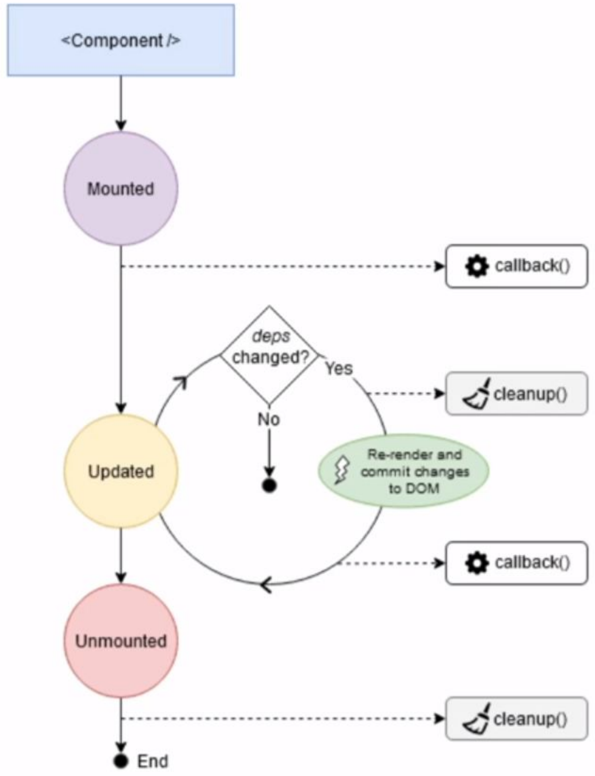
The function returned from the useEffect hook is invoked by React while unmounting the component & while cleaning up effects from previous renders hence loosely mimicking componentWillUnmount()

👉 empty dependency array

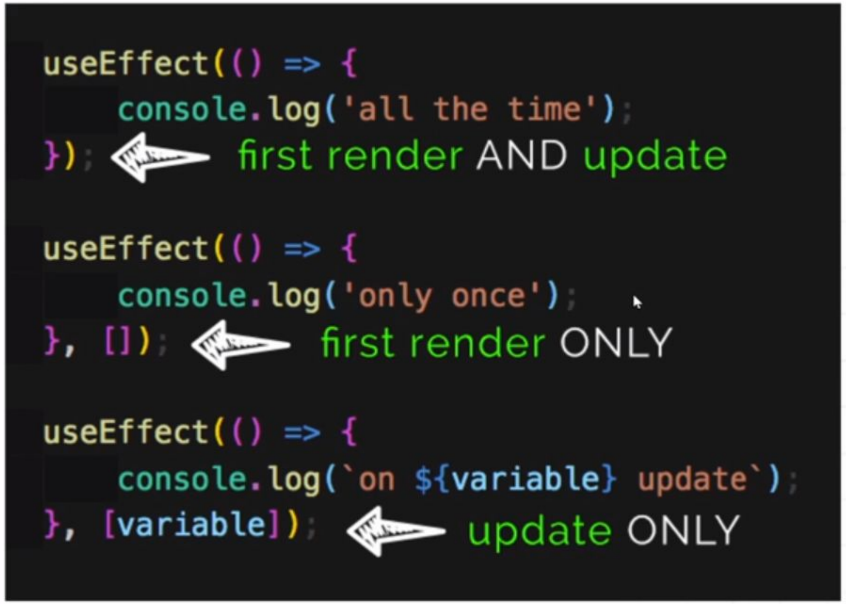
Means that the effect() function will get called only once hence replicating componentDidMount()

🧐

How does useEffect work?



useEffect – dependency? / no dependency?



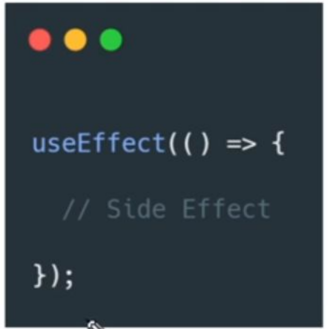
```
useEffect(() => {
    console.log('all the time');
});
```

```
useEffect(() => {
    console.log('only once');
}, []);
```

```
useEffect(() => {
    console.log(`on ${variable} update`);
}, [variable]);
```

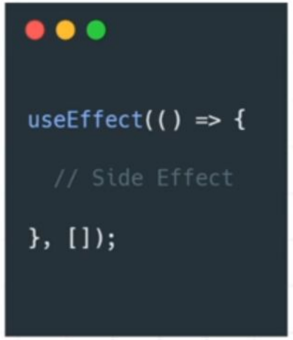
Different types of dependency in useEffect

1. Side Effect Runs After Every Render



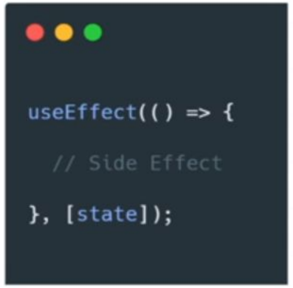
```
useEffect(() => {
  // Side Effect
});
```

2. Side Effect Runs Only Once After Initial Render



```
useEffect(() => {
  // Side Effect
}, []);
```

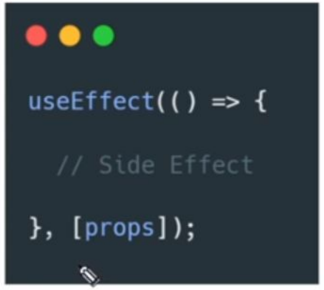
3. Side Effect Runs After State Value Changes



```
useEffect(() => {
  // Side Effect
}, [state]);
```

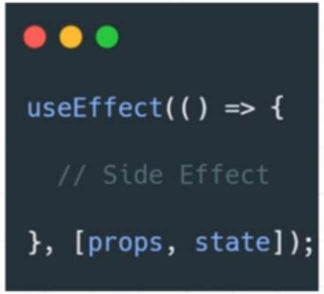
Different types of dependency in useEffect

4. Side Effect Runs After Props Value Change



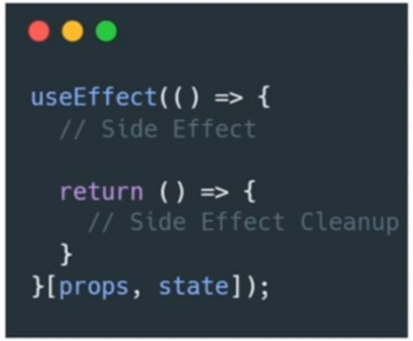
```
useEffect(() => {

  // Side Effect

}, [props]);
```

5. Side Effect Runs After Props or State Value Change



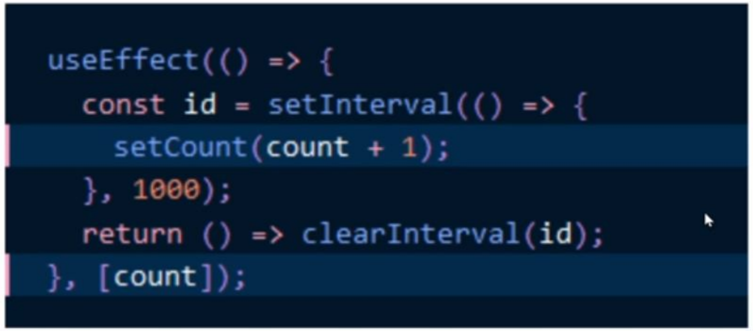
```
useEffect(() => {

  // Side Effect

}, [props, state]);
```

6. Side EffectCleanup

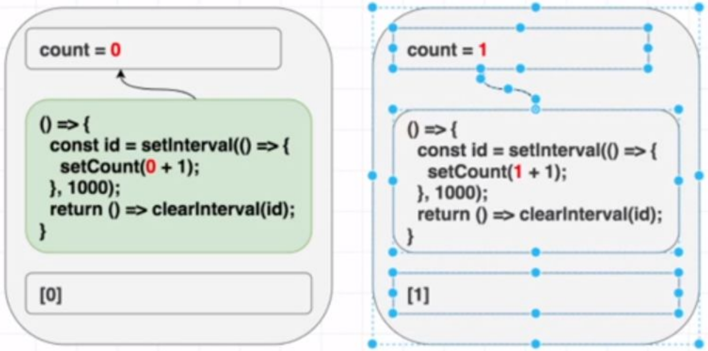


```
useEffect(() => {
  // Side Effect

  return () => {
    // Side Effect Cleanup
  }
}[props, state]);
```

useEffect() with setInterval()



```
useEffect(() => {
  const id = setInterval(() => {
    setCount(count + 1);
  }, 1000);
  return () => clearInterval(id);
}, [count]);
```



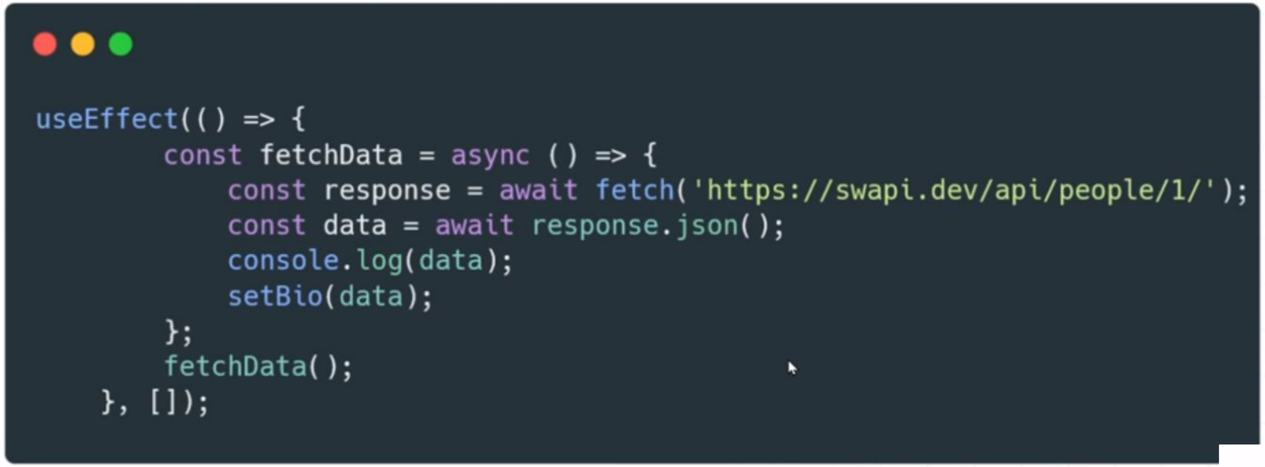
```
count = 0

() => {
  const id = setInterval(() => {
    setCount(0 + 1);
  }, 1000);
  return () => clearInterval(id);
}

[0]
```

```
count = 1

() => {
  const id = setInterval(() => {
    setCount(1 + 1);
  }, 1000);
  return () => clearInterval(id);
}

[1]
```

Use cases of useEffect()

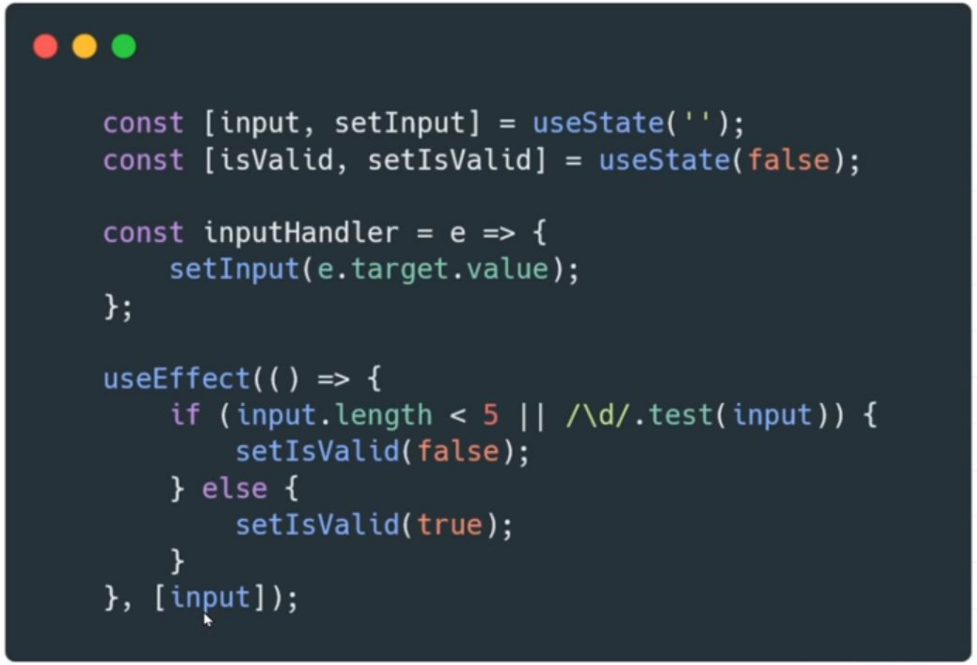
1. Running once on mount: fetch API data
2. Running on state change: validating input field
3. Running on state change: manipulation dom directly

1. Running once on mount: fetch API data



```
useEffect(() => {
        const fetchData = async () => {
            const response = await fetch('https://swapi.dev/api/people/1/');
            const data = await response.json();
            console.log(data);
            setBio(data);
        };
        fetchData();
    }, []);
```

2. Running on state change: validating input field



```
const [input, setInput] = useState('');
const [isValid, setIsValid] = useState(false);

const inputHandler = e => {
    setInput(e.target.value);
};

useEffect(() => {
    if (input.length < 5 || /\d/.test(input)) {
        setIsValid(false);
    } else {
        setIsValid(true);
    }
}, [input]);
```

Thank you!