A Warning

- We know `useState`, `useId`, `useRef` hooks
    - `useEffect` is another hook

Warning

- `useEffect` is the gnarliest piece of React
- Common source of mistakes
- Not needed for MOST things
- But comes up often
- Often used in Interviews to test

useEffect Hook

What does useEffect do?

- `useEffect()` is passed a callback
- callback runs *after* the component renders

`useEffect()` used to create a **side-effect** of rendering

What is a Side Effect?

- **A change outside of the returned value**
    - Function does X, but also causes Y
- Side-effects usually bad (complex, unexpected)
    - But needed to work with external systems
    - Update screen, talk with database, etc
    - For us: Anything outside Components
        - Examples coming

Basic example

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

function App() {

  useEffect( () => console.log('in effect') );

  console.log('in app');

  return (
    <>
      App
    </>
  );
}
```

Console

```
in app
in app (greyed out?)
in effect
in effect
```

Why is Console Showing Messages Twice?

React 18 added a feature

- In "development mode"
  - The dev server via `npm run dev`
- Components rendered a second time
  - Largely to highlight `useEffect` problems

Not itself a problem

- Won't happen in production
  - The built files using `npm run build`
- Exists to reveal problems!

useEffect callback called on every rerender

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

function App() {

  const [ count, setCount ] = useState(0);

  useEffect( () => console.log('in effect') );

  console.log('in app');

  return (
    <>
      <button onClick={ () => setCount(count+1) }>
        {count}
      </button>
    </>
  );
}
```

Each (initial) `in app` followed by an `in effect`

useEffect dependency array

useEffect callback doesn't have to run on ALL renders

- Can be passed a second argument
  - **dependency array**
  - List of values
- Effect callback runs on render if any changed

Dependency Array Demonstration

```
function App() {
  const [ count, setCount ] = useState(0);
  const [ watched, setWatched ] = useState(0);

  useEffect(
    () => console.log('in effect'),
    [ watched ],
  );

  console.log('in app');

  return (
    <>
      <button onClick={ () => setCount(count+1) }>
        Unwatched: {count}
      </button>
      <button onClick={ () => setWatched(watched+1) }>
        Watched: { watched }
      </button>
    </>
  );
}
```

Simple Results

- Whenever the `watched` value changed
    - `<App>` rerendered
    - useEffect callback was called
- When `count` (unwatched) value changed
    - `<App>` rerendered
    - useEffect callback NOT called

Changing state every render is an Infinite Loop

```
const [state, setState] = useState(0);

useEffect(
  () => setState(state + 1),
  [state],
);
```

- On render, run effect
- Effect changes state, triggers render
- = **infinite loop**

useEffect callback

- Should NOT change state
- OR only *conditionally* change the state

What if empty deps array?

What if:

```
useEffect(
  () => console.log('in effect'),
  [],
);
```

Empty dependency array results

- useEffect callback runs on first render
    - Not on any later renders
    - No value in array has changed!
- If component is removed from page and reapplied
    - callback once again runs on first render
- If multiple instances of component
    - callback runs on first render of each instance

When to use dependency array

First questions:

- What is your "effect"?
- Why are you doing so based on render?

Component will re-render each time state changes

- Do you want your effect each time state changes?

If your effect is based on 1+ values

- Those values are your dependency array

Effect: Increasing Counter

Hint: **Basis of many interview questions**

- Because it isn't just a "it works" question

A Component that shows a Counter

- Counter starts when component FIRST renders
- Automatically increments (roughly 1/second)
- Cleans up when component removed

Creating the increment is an "effect"

Component Base Structure

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

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div className="counter">{count}</div>
  );
}

export default Counter;
```

Increase `count` ~1/second

```
const [count, setCount] = useState(0);

useEffect(
  () => {
    setInterval( () => {
      console.log('incrementing', count);
      setCount(count + 1);
    }, 1000);
  }
);
```

But this has a problem!

Too Many Effects

The interval was changing state (using `setCount()`)

- Which triggers a rerender
- Each render added a NEW effect
    - An additional Interval
        - Each triggering more renders

We only want to create the interval once

- Use a dependency array

Why the smaller counts?

- Explosion of intervals wasn't the only issue
- We also saw smaller numbers later on - why?
- `setCount(count + 1)` uses original `count`
    - `count` is never changed
    - Later renders have separate `count` variables
- `setCount(count => count + 1)` fixes that
- Same as `setCount(current => current + 1)`

Adding the dependency array

```
useEffect(
  () => {
    setInterval( () => {
      console.log('incrementing');
      setCount(current => current + 1);
    }, 1000);
  },
  [] // empty = effect on first render only
);
```

We DO NOT want `count` as a dependency

- It changes = infinite loop
- Using the function form for setter works fine
- Leaving `count` out will generate a warning
    - Unless we use pass a function to the setter

Why is counter going up by 2?

This is because of that development feature

- Our effect is running twice

Why would they mess us up like this?

- Actually a sign of a problem in our code
- Let's look at that problem first
    - Then consider why double render helped

We still have a problem

`<Counter>` "works"

- Except for double count
- What happens when removed from page?

```
function App() {
  const [showCounter, setShowCounter] = useState(false);
  return (
    <>
      <button onClick={ () => {
        setShowCounter( !showCounter )
      }}>
        Toggle Counter
      </button>
      { showCounter && <Counter/> }
    </>
  );
}
```

Interval from effect still exists

Interval exists even after component is removed

- Adding component back creates extra effect

This is why our count was upping by 2

- Initial effect was run twice
- Two Intervals, both raising the same state

We need to "clean up" our effect

useEffect callback can return a function

The returned function is used for "cleanup"

React automatically calls cleanup function:

- When Component removed from page
- Just before useEffect callback called again

Example: If effect created timeouts or intervals

- Remove them when component(+state) removed
- Remove before creating new timeouts/intervals

useEffect cleanup function

```
useEffect(
  () => {

    console.log('in effect', count);

    return () => {
      console.log('cleanup', count);
    };
  },
  [],
);
```

Cleanup Counter

- To remove interval we need `intervalId`
  - But we don't want it in state (rerenders)
  - We use a **closure**
    - Reference to variable no longer in scope

```
useEffect(
  () => {
    const intervalId = setInterval( () => {
      console.log('incrementing');
      setCount(current => current + 1);
    }, 1000);
    return () => {
      console.log('cleanup');
      clearInterval(intervalId);
    };
  },
  [] // empty = effect on first render only
);
```

Clean!

- "cleanup" in the console
- Only counts by 1!
- Stops when component removed

Second render made problem more noticeable!

Effects not tied to component lifecycle cause issues

- Be sure to have cleanup for lasting effects
- Consider if component may no longer be there
    - For async effects
- Use the double-render in dev as a "canary"

What is a Canary

From "Canary in a coal mine"

- Miners would take a caged bird with them
- Bird would show signs of bad air before humans
- Humans could leave before passing out/dying
    - Hopefully WITH the bird

Practices that help reveal problems early:

- **Canary**

Summary - useEffect

A hook that takes a callback

- Callback runs after component renders
- Used for "side effects" to render
    - setup/cleanup needed for component

Changing state in effect can cause infinite loop

- Think about it before changing state

Summary - Dependency Array

Second param to useEffect is a **dependency array**

- If not present
  - Callback runs every render
- If present but empty (`[]`)
  - Callback runs after first render only
- If present with values
  - Callback runs if any values change
- If calling a state setter (avoid infinite loop!)
  - Pass setter function
  - Don't reference changing state

Summary - Cleanup function

The useEffect callback can return a function

- Automatically used for **cleanup**
- Remove timeouts/intervals
- Disconnect any external effects

Cleanup runs

- Before useEffect callback runs again
- When component removed (unmounted)

Summary - Double Render in dev

React 18 does a double render in development

- Can reveal when effects aren't being cleaned up
- Only useful if you pay attention
    - Keep console clean
    - Deal with warnings and errors
    - Check console often