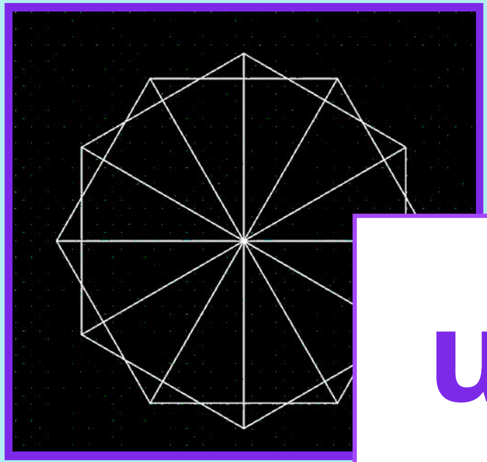
useEffect

Presented by
Tom Isles

Canva



Motivation

A few weeks ago, we created a component to display the date and time when we loaded the page.

Today, we'll track time in a different way.
We'll be making a stopwatch.

Demo

Methods

1) Outside the Component
2) With useState
3) With useEffect

```
let seconds = 0;

window.setInterval(() => {
   seconds += 1;
}, 1000);

const reset = () => {
   seconds = 0;
}

function App() {
   return (
      <section className="App">
         <div className="Seconds">
            <p>Seconds Passed:</p>
            <p>{seconds}</p>
         </div>
         <button onClick={reset} />
      </section>
   );
}
```

Methods

1) Outside the Component X
2) With useState
3) With useEffect

```
let seconds = 0;

window.setInterval(() => {
  seconds += 1;
}, 1000);

const reset = () => {
  seconds = 0;
}

function App() {
  return (
    <section className="App">
      <div className="Seconds">
        <p>Seconds Passed:</p>
        <p>{seconds}</p>
      </div>
      <button onClick={reset} />
    </section>
  );
}
```

Methods

1) Outside the Component X

2) With useState

3) With useEffect

```
function App() {
   const [seconds, setSeconds] = React.useState(0);

   window.setInterval(() => {
      setSeconds(seconds + 1);
   }, 1000);

   const reset = () => setSeconds(0);

   return (
      <section className="App">
         <div className="Seconds">
            <p>Seconds Passed:</p>
            <p>{seconds}</p>
         </div>
         <button onClick={reset} />
      </section>
   );
}
```

Methods

1) Outside the Component X
2) With useState X
3) With useEffect

```
function App() {
   const [seconds, setSeconds] = React.useState(0);

   window.setInterval(() => {
      setSeconds(seconds + 1);
   }, 1000);

   const reset = () => setSeconds(0);

   return (
      <section className="App">
         <div className="Seconds">
            <p>Seconds Passed:</p>
            <p>{seconds}</p>
         </div>
         <button onClick={reset} />
      </section>
   );
}
```

Methods

1) Outside the Component X
2) With useState X
3) With useEffect

```
function App() {
   const [seconds, setSeconds] = React.useState(0);

   React.useEffect(() => {
      return window.setInterval(() => {
         setSeconds(s => s + 1);
      }, 1000);
   }, []);

   const reset = () => setSeconds(0);

   return (
      <section className="App">
         <div className="Seconds">
            <p>Seconds Passed:</p>
            <p>{seconds}</p>
         </div>
         <button onClick={reset} />
      </section>
   );
}
```

What is useEffect?

useEffect is a React hook. You have learned about useState already, which is another example of a React hook.

useEffect reacts to changes in our component. It lets us pass in a function. The function we pass in will be triggered if certain conditions are met within the component.

This allows us to react to changes in the component with custom code.

It is a substitute for react class component methods like **componentDidMount** and **componentShouldUpdate**.

Building a useEffect hook

The first parameter of React.useEffect is the function that you would like to run.

In this case, we're using *window* to set an interval. The function inside the interval will run every 1000 milliseconds, give or take.

```
React.useEffect(() => {
   window.setInterval(() => {

   }, 1000);
});
```

Building a useEffect hook

The first parameter of React.useEffect is the function that you would like to run.

In this case, we're using *window* to set an interval. The function inside the interval will run every 1000 milliseconds, give or take.

```
React.useEffect(() => {
   window.setInterval(() => {

   }, 1000);
});
```

The interval is defined inside useEffect

Building a useEffect hook

We can mix and match hooks to give us a lot of power.

In this example, we're using a state hook to store the number of seconds that have passed, and we're updating it inside the interval to increment it by 1, using *setSeconds*.

```
const [seconds, setSeconds] = React.useState(0);

React.useEffect(() => {
   window.setInterval(() => {
      setSeconds(s => s + 1);
   }, 1000);
});
```

Building a useEffect hook

We can mix and match hooks to give us a lot of power.

In this example, we're using a state hook to store the number of seconds that have passed, and we're updating it inside the interval to increment it by 1, using *setSeconds*.

```
const [seconds, setSeconds] = React.useState(0);

React.useEffect(() => {
   window.setInterval(() => {
      setSeconds(s => s + 1);
   }, 1000);
});
```

Inside the interval, we call our state setter

Demo

Building a useEffect hook

The second parameter in useEffect is the dependency array. In the array, we provide a list of props and/or state objects.

If any of the references defined in the dependency array are mutated, then the function inside useEffect will be triggered. If the component renders but the props and state in the array have *not* been mutated, the function will not trigger.

```
const [seconds, setSeconds] = React.useState(0);

React.useEffect(() => {
   window.setInterval(() => {
      setSeconds(s => s + 1);
   }, 1000);
}, []);
```

Building a useEffect hook

The second parameter in useEffect is the dependency array. In the array, we provide a list of props and/or state objects.

If any of the references defined in the dependency array are mutated, then the function inside useEffect will be triggered. If the component renders but the props and state in the array have *not* been mutated, the function will not trigger.

```
const [seconds, setSeconds] = React.useState(0);

React.useEffect(() => {
   window.setInterval(() => {
      setSeconds(s => s + 1);
   }, 1000);
}, []);
```

The dependency array specifies which props / state variables will trigger the effect hook if they get changed.

Building a useEffect hook

If no dependency array is provided, then the useEffect function will be triggered after *every* render.

If an empty array is provided, the use Effect function will only run once, after the component first renders. Then, nothing can trigger it again.

```
const [seconds, setSeconds] = React.useState(0);

React.useEffect(() => {
   window.setInterval(() => {
      setSeconds(s => s + 1);
   }, 1000);
}, []);
```

Building a useEffect hook

If no dependency array is provided, then the useEffect function will be triggered after *every* render.

If an empty array is provided, the use Effect function will only run once, after the component first renders. Then, nothing can trigger it again.

```
const [seconds, setSeconds] = React.useState(0);

React.useEffect(() => {
   window.setInterval(() => {
      setSeconds(s => s + 1);
   }, 1000);
}, []);
```

The interval here is set only after the first render, and then the function is never run again.

Building a useEffect hook

In the useEffect function we pass in, we can *return* a function from it.

The function that we return is a **cleanup function.**

If the useEffect function is triggered later, the cleanup function will run immediately prior to the trigger.

When the component is "unmounted" - removed from the current view - this cleanup function will also run.

```
const [seconds, setSeconds] = React.useState(0);

React.useEffect(() => {
   const interval = window.setInterval(() => {
      setSeconds(s => s + 1);
   }, 1000);

   return () => clearInterval(interval);
}, []);
```

Building a useEffect hook

In the useEffect function we pass in, we can *return* a function from it.

The function that we return is a **cleanup function.**

If the useEffect function is triggered later, the cleanup function will run immediately prior to the trigger.

When the component is "unmounted" - removed from the current view - this cleanup function will also run.

```
const [seconds, setSeconds] = React.useState(0);

React.useEffect(() => {
   const interval = window.setInterval(() => {
      setSeconds(s => s + 1);
   }, 1000);

   return () => clearInterval(interval);
}, []);
```

By returning a function that calls clearInterval on our interval, we ensure that only one interval is ever defined at any one time.

Building a useEffect hook

One of the most common use cases for useEffect is to set up subscriptions. In this case, we're subscribing to a time interval.

Cleanup allows us to *unsubscribe* when the subscription is no longer relevant.

```
const [seconds, setSeconds] = React.useState(0);

React.useEffect(() => {
   const interval = window.setInterval(() => {
      setSeconds(s => s + 1);
   }, 1000);

   return () => clearInterval(interval);
}, []);
```

By returning a function that calls clearInterval on our interval, we ensure that only one interval is ever defined at any one time.

Demo

Great Work!

We now have an application that counts the number of seconds that have passed. **Now, we will build on this by also storing the number of minutes that have passed.**

```
// We'll need some state
const [minutes, setMinutes] = React.useState(0);
```

Tracking our Minutes

We'll need to add another effect hook in order to track our minutes.

```
function App() {
   const [seconds, setSeconds] = React.useState(0);
   const [minutes, setMinutes] = React.useState(0);

   React.useEffect(() => {
      return window.setInterval(() => {
         setSeconds(s => s + 1);
      }, 1000);
   }, []);

   const reset = () => {
      setSeconds(0);
      setMinutes(0);
   }

   return (
      <section className="App">
         <div className="Seconds">
            <p>Time Passed:</p>
            <p>{minutes}:{seconds}</p>
         </div>
         <button onClick={reset} />
      </section>
   );
}
```

Tracking our Minutes

We'll need to add another effect hook in order to track our minutes.

```
function App() {
   const [seconds, setSeconds] = React.useState(0);
   const [minutes, setMinutes] = React.useState(0);

   React.useEffect(() => {
      return window.setInterval(() => {
         setSeconds(s => s + 1);
      }, 1000);
   }, []);

   React.useEffect(() => {
      if (seconds >= 60) {
         setSeconds(0);
         setMinutes(m => m + 1);
      }
   }, [seconds]);

   const reset = () => {
      setSeconds(0);
      setMinutes(0);
   }

   return (
      <section className="App">
         <div className="Seconds">
            <p>Time Passed:</p>
            <p>{minutes}:{seconds}</p>
         </div>
         <button onClick={reset} />
      </section>
   );
}
```

Tracking our Minutes

We've added multiple useEffect hooks into one component.

If two useEffect hooks are defined, then it is possible that then they will both need to trigger. If that's the case, **they will trigger in the order that they were declared.**

Both hooks will trigger after our first render, and the second hook will trigger on any subsequent change to the *seconds* parameter.

```
function App() {
   const [seconds, setSeconds] = React.useState(0);
   const [minutes, setMinutes] = React.useState(0);

   React.useEffect(() => {
      return window.setInterval(() => {
         setSeconds(s => s + 1);
      }, 1000);
   }, []);

   React.useEffect(() => {
      if (seconds >= 60) {
         setSeconds(0);
         setMinutes(m => m + 1);
      }
   }, [seconds]);

   const reset = () => {
      setSeconds(0);
      setMinutes(0);
   }

   return (
      <section className="App">
         <div className="Seconds">
            <p>Time Passed:</p>
            <p>{minutes}:{seconds}</p>
         </div>
         <button onClick={reset} />
      </section>
   );
}
```

Tracking our Minutes

We've added multiple useEffect hooks into one component here.

If two useEffect hooks are defined, then it is possible that then they will both need to trigger. If that's the case, they will trigger in the order that they were declared.

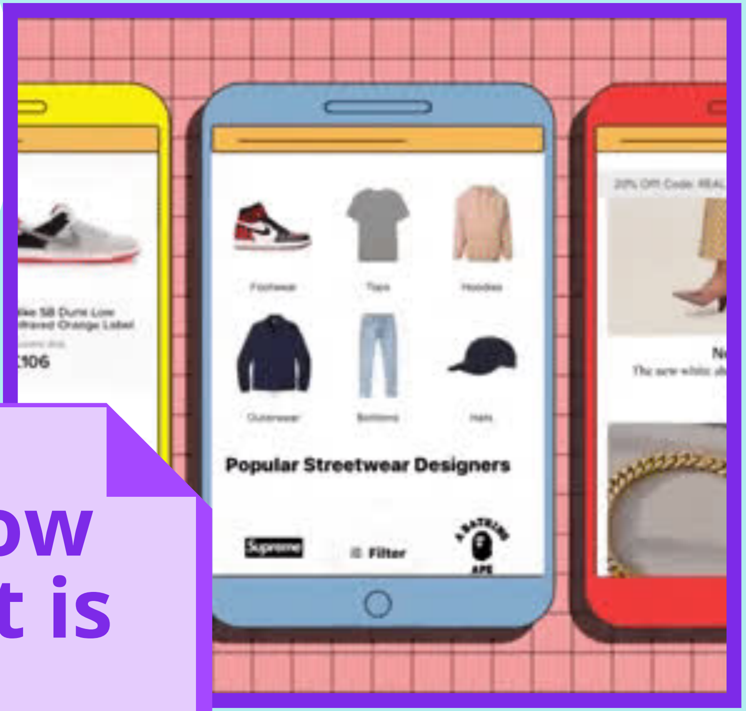
Both hooks will trigger after our first render, and the second hook will trigger on any subsequent change to the *seconds* parameter.

```
function App() {
   const [seconds, setSeconds] = React.useState(0);
   const [minutes, setMinutes] = React.useState(0);

   React.useEffect(() => {
      return window.setInterval(() => {
         setSeconds(s => s + 1);
      }, 1000);
   }, []);

   React.useEffect(() => {
      if (seconds >= 60) {
         setSeconds(0);
         setMinutes(m => m + 1);
      }
   }, [seconds]);

   const reset = () => {
      setSeconds(0);
      setMinutes(0);
   }

   return (
      <section className="App">
         <div className="Seconds">
            <p>Time Passed:</p>
            <p>{minutes}:{seconds}</p>
         </div>
         <button onClick={reset} />
      </section>
   );
}
```

Passing the seconds state into the dependency array tells the effect to trigger whenever seconds is mutated

Demo

Q



How do we know when useEffect is triggered?

Recap: When does rendering occur?

</>

What causes a React component to render?

- If the component's props are changed.
- If the component's state is changed.
- If a re-render occurs above the component in the tree, and the component is not memoized.

When is useEffect triggered?

At what point does the useEffect callback get called?

The useEffect function will be called if one of the dependencies in its dependency array is mutated, or, if no array is provided, when every render occurs.

The function itself is always called asynchronously, after a component has rendered.

Why useEffect?

Why do we have to use useEffect?

Determinism is an important property in our function components. That means, given identical props and state, the component should return the same JSX every time. We refer to functions with this property as **pure** functions.

This has important implications for the correctness & maintainability of React codebases.

Why useEffect?

Why do we have to use useEffect?

In order for our functions to be deterministic, **all of our state must be tracked by React.** If React needed to track variables outside of components, then we could not guarantee that our function components are deterministic, because those variables lay outside our components and may change at any time.

Why useEffect?

This is why we have **useState**. It allows us to store variables inside the React render loop.

Similarly, **useEffect** allows us to define arbitrary functions to be executed, so we can execute custom code.

Common Use Cases

- Fetching Data.
- Managing Subscriptions.
- Setting up Timers.
- Performing native DOM manipulations when needed.

Demo

Some More Gotchas

The Dependency Array
Any pieces of state we consume, ESLint will automatically add to our dependency array.
We need to be very careful about what state we consume. Consider the following:

```
setSeconds(seconds + 1)
```

seconds would form part of the dependency array, creating an infinite loop.
So, we do the following:

```
setSeconds(seconds => seconds + 1)
```

Some More Gotchas

After Render
The effect function is triggered after first render. This can bite you if you're expecting your function to be called before your component renders.

useLayoutEffect
Same hook as useEffect, but instead of triggering after render, it triggers after the DOM mutates, but before render.

Good Work!