React Effects and Refs

Download Demo Code

Goals

- Describe what the ***useEffect*** hook does
- Use effects to run code after the rendering phase of a component
- Describe what the ***useRef*** hook does
- Use refs to access DOM nodes and work with timers

useEffect

- React comes with a built in hook for "side effects"
- Fetching data, starting a timer, and manually changing the DOM are all side effects
- Each render has its own effects
- Sometimes these effects require clean-up (clearing a timeout, closing a connection)

- ***useEffect*** will run after the first render
- ***useEffect*** will run after all rerenders by default
- ***useEffect*** accepts a callback function as its first argument
- ***useEffect*** returns undefined or a function
  - If you return a function, the function will be run before the component unmounts or before the effect runs again

demo/counter-effects/src/EffectExample.js

```
import React, { useState, useEffect } from "react";

function EffectExample() {
  const [num, setNum] = useState(0);

  function increment(evt) {
    setNum(n => n + 1);
  };

  useEffect(function setTitleOnRerender() {
    document.title = `WOW!${"!".repeat(num)}`;
  });

  return (
    <div>
      Let's get excited.
      <button onClick={increment}>Get more excited.</button>
    </div>
  );
}
```

useEffect arguments

2nd argument to useEffect

- You can tell React to skip applying an effect if certain values haven't changed between re-renders.
- ***useEffect*** accepts an array as its second argument where you place these values (also called dependencies)
- What you pass to the array can help avoid performance issues (we'll talk about this more later)

If you want to run an effect and clean it up only once (on mount and unmount), you can pass an empty array ([]) as a second argument.

This tells React that your effect doesn't depend on any values from props or state, so it never needs to re-run.

Be careful about using this pattern when your effect *does* depend on props or state, as React will give you a warning.

Fetching Data on Initial Render

A Typical Use Case for *useEffect*

- It's very common that when a component renders, you'll want to fetch some data from an external data source or API
- We want to do this after the component first renders so that we can show the user something (e.g. a loading screen) while we fetch that data
- To fetch correctly, we'll run an effect that only happens once and when the API call is finished, we'll set our state and render the component again

demo/github-profile-viewer/src/ProfileViewer.js

```
import React, { useState, useEffect } from "react";
import axios from "axios";

/** GitHub Profile Component --- shows info from GH API */

function ProfileViewer() {
  const [profile, setProfile] = useState(null);

  // this is called *after* component first added to DOM
  useEffect(function fetchUserWhenMounted() {
    async function fetchUser() {
      const userResult = await axios.get(
        "https://api.github.com/users/elie");
      setProfile(userResult.data);
    }
    fetchUser();
  }, []);

  return (
    <div>{profile ? <h2>{profile.name}</h2> : <i>{loading}</i>}</div>
  );
};
```

Some important notes here:

- useEffect cannot be an async function, we must define an async function inside and invoke it
- make sure that we change state after getting back a response
- don't forget to handle errors correctly!

Updating After Subsequent Renders

Fetching Data Later

Goal: fetch data not after the first render, but after a later state change

Example: Text Search

demo/github-profile-viewer/src/ProfileViewerWithSearch.js

```
import React, { useState, useEffect } from "react";
import axios from "axios";
import ProfileSearchForm from "./ProfileSearchForm";

const BASE_URL = "https://api.github.com/users";

/** GitHub Profile Component --- shows info from GH API */

function ProfileViewerWithSearch() {
  const [profile, setProfile] = useState(null);
  const [username, setUsername] = useState("elie");

  function search(username) {
    setUsername(username);
  };

  // this is called after component is first added to DOM
  // and every time username changes
  useEffect(function fetchUserOnUsernameChange() {
    async function fetchUser() {
      const userResult = await axios.get(`${BASE_URL}/${username}`);
      setProfile(userResult.data);
    }
    fetchUser();
  }, [username]);


  return (
    <div>
      <ProfileSearchForm search={search} />
      {profile ? <h2>{profile.name}</h2> : <i>{loading}</i>}
    </div>
  );
};

export default ProfileViewerWithSearch;
```

Cleaning up an Effect

In our previous example, we only fetched data on mount and on update, but it's very common to handle events when the component will be removed from the DOM. Some common examples include

- clearing intervals or timeouts
- removing an event listener
- unsubscribing
- disconnecting from a socket

Cleanup with useEffect

To do this, we return a function from useEffect!

```
useEffect(() => {
  // runs on the first render and all times after
  // because we didn't pass in an array as a 2nd arg!
  console.log("Effect ran!");

  // if we return a function
  // it will run when the component unmounts
  // or before the effect runs again
  return () => console.log("In the cleanup phase!");
})
```

useRef

- ***useRef*** is another built-in hook in React.
- It returns a mutable object with a key of ***current***, whose value is equal to the initial value passed into the hook.
- The object persists across renders (so it's like a local variable, but independent of state).
- Mutating the object does not trigger a re-render.

Common Applications of useRef

1. Accessing an underlying DOM element
2. Setting up / clearing timers

Accessing the DOM

Sometimes, you need to access an HTMLElement to make use of a Web API or to integrate some other JavaScript library.

This is a great time to use a ref!

Accessing the DOM Example

demo/refs-app/src/Video.js

```
import React, { useState, useRef, useEffect } from "react";

function Video({
  src = "https://media.giphy.com/media/KctGIT2JHvVRC7ESeR/giphy.mp4"
}) {
  const video = useRef();
  const [speed, setSpeed] = useState(1);

  useEffect(function adjustPlaybackRateOnVideo() {
    // video.current is the video HTML Element
    // video elements have a .playbackRate
    // that allow you to speed up / slow down a video
    video.current.playbackRate = speed;
  }, [video, speed]);

  return (
    <div>
      <video muted autoPlay loop ref={video}>
        <source src={src} />
      </video>
      <div>
        <button onClick={evt => setSpeed(s => s / 2)}>slow</button>
        <button onClick={evt => setSpeed(s => s * 2)}>fast</button>
        <p>Current speed: {speed}x</p>
      </div>
    </div>
  );
}
```

- ***.playbackRate*** can only be changed if you have access to the underlying HTML element.
- A ref can get us access to the DOM element!
- To assign a ref to a DOM element, use the ***ref*** attribute on the desired element.

Timers

Another great time to use a ref is with timers like ***setInterval***.

setInterval returns a timer ID, which we need to stop the ***setInterval*** from running.

We can store that ID in a ref, and then stop the timer when the component is removed from the DOM.

Timer Example

demo/refs-app/src/TimerWithRef.js

```
import React, { useState, useEffect, useRef } from "react";

function TimerWithRef() {
  const timerId = useRef();
  let [count, setCount] = useState(0);

  useEffect(function setCounter() {
    console.log("EFFECT RAN!");
    timerId.current = setInterval(() => {
      setCount(c => c + 1);
    }, 1000);

    return function cleanUpClearTimer() {
      console.log("Unmount ID", timerId.current);
      clearInterval(timerId.current);
    };
  }, [timerId]);

  return (
    <div>
      <h1>{count}</h1>
      <p>Timer id is {timerId.current}.</p>
    </div>
  );
}
```

Antipattern for useRef

Since refs can expose DOM elements for us, it can be tempting to use them to access the DOM and make changes (toggle classes, set text, etc).

This is **not** how refs should be used. React should control the state of the DOM!

From the docs:

> *Your first inclination may be to use refs to "make things happen" in your app. If this is the case, take a moment and think more critically about where state should be owned in the component hierarchy.*