React Lecture - 18

forwardRef and Effects

Introduction

When you put a ref on a built-in component that outputs a browser element like <input />, React will set that ref's current property to the corresponding DOM node (such as the actual <input /> in the browser).

However, if you try to put a ref on your own component, like <MyInput />, by default you will get null.

Introduction

This happens because by default React does not let a component access the DOM nodes of other components. Not even for its own children! This is intentional. Refs are an escape hatch that should be used sparingly. Manually manipulating another component's DOM nodes makes your code even more fragile.

Instead, components that want to expose their DOM nodes have to opt in to that behavior. A component can specify that it "forwards" its ref to one of its children. Here's how MyInput can use the forwardRef API

forwardRef

This is how it works:

```
const MyInput = forwardRef((props, ref) => {
  return <input {...props} ref={ref} />;
});
```

<MyInput ref={inputRef} /> tells React to put the corresponding DOM node into inputRef.current. However, it's up to the MyInput component to opt into that—by default, it doesn't.

The MyInput component is declared using forwardRef. This opts it into receiving the inputRef from above as the second ref argument which is declared after props.

MyInput itself passes the ref it received to the <input> inside of it.

Effects

Introduction

Some components need to synchronize with external systems.

For example, you might want to control a non-React component based on the React state, set up a server connection, or send an analytics log when a component appears on the screen.

Effects let you run some code after rendering so that you can synchronize your component with some system outside of React.

What are Effects and how are they different from events?

Till now we have studied two types of logics in React Components.

1. Rendering Logic
2. Event Handlers

Sometimes these are not enough.

Consider a ChatRoom component that must connect to the chat server whenever it's visible on the screen.

Connecting to a server is not a pure calculation (it's a side effect) so it can't happen during rendering. However, there is no single particular event like a click that causes ChatRoom to be displayed.

What are Effects and how are they different from events?

Effects let you specify side effects that are caused by rendering itself, rather than by a particular event.

Sending a message in the chat is an event because it is directly caused by the user clicking a specific button

However, setting up a server connection is an Effect because it should happen no matter which interaction caused the component to appear.

Effects run at the end of a commit after the screen updates. This is a good time to synchronize the React components with some external system (like network or a third-party library).

How to write Effect ?

To write an Effect, follow these three steps:

1. **Declare an Effect**.

   By default, your Effect will run after every render.

2. **Specify the Effect dependencies**.

   Most Effects should only re-run when needed rather than after every render. For example, a fade-in animation should only trigger when a component appears. Connecting and disconnecting to a chat room should only happen when the component appears and disappears, or when the chat room changes. You will learn how to control this by specifying dependencies.

3. **Add cleanup if needed**.

   Some Effects need to specify how to stop, undo, or clean up whatever they were doing. For example, “connect” needs “disconnect”, “subscribe” needs “unsubscribe”, and “fetch” needs either “cancel” or “ignore”. You will learn how to do this by returning a cleanup function.

Step 1: Declare an Effect

To declare an Effect in your component, import the useEffect Hook from React:

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

Then, call it at the top level of your component and put some code inside your Effect:

```
function MyComponent() {
  useEffect(() => {
    // Code here will run after *every* render
  });
  return <div />;
}
```

Every time your component renders, React will update the screen and then run the code inside useEffect. In other words, useEffect “delays” a piece of code from running until that render is reflected on the screen.

Step 2: Specify the effect dependency

By default, Effects run after every render. Often, this is not what you want:

- Sometimes, it's slow. Synchronizing with an external system is not always instant, so you might want to skip doing it unless it's necessary. For example, you don't want to reconnect to the chat server on every keystroke.
- Sometimes, it's wrong. For example, you don't want to trigger a component fade-in animation on every keystroke. The animation should only play once when the component appears for the first time.

You can tell React to skip unnecessarily re-running the Effect by specifying an array of dependencies as the second argument to the useEffect call. Start by adding an empty [] array

Step 2: Specify the effect dependency

The dependency array can contain multiple dependencies. React will only skip re-running the Effect if all of the dependencies you specify have exactly the same values as they had during the previous render. React compares the dependency values using the Object.is comparison.

Step 3: Add Cleanup if needed (OPTIONAL)

When we return a function from the useEffect callback function, it is called a cleanup function

React will call your cleanup function each time before the Effect runs again, and one final time when the component unmounts (gets removed).

```
useEffect(() => {
  const connection = createConnection();
  connection.connect();
  return () => {
    connection.disconnect();
  };
}, []);
```