Exercise 13: React Hook (useEffect)

Objectives and Outcomes

useEffect is a React hook that allows you to perform side effects in functional components. By using the useEffect hook, you can handle side effects in a declarative way within functional components. It provides a flexible and efficient mechanism to perform actions that are not directly related to rendering the UI.

Exercises

1. Data Fetching

Create a UserPosts component that fetches and displays a list of posts for a given user ID using the JSONPlaceholder API ([https://jsonplaceholder.typicode.com/](https://jsonplaceholder.typicode.com/guide/)). The component should fetch data when it mounts and update the data when the user ID prop changes

**Expectations**

* Fetch data from the JSONPlaceholder API.
* Display a list of posts for the given user ID.
* Refetch data when the user ID prop changes.

**Hint:**

Initialize state variable posts with an empty array: We create a state variable posts using the useState hook and set its initial value to an empty array.

const [posts, setPosts] = useState([]);

Create an async function to fetch data and update the posts state: We use the useEffect hook to fetch data when the component mounts and when the userId prop changes. We create an async function fetchData inside the useEffect hook to fetch data from the JSONPlaceholder API using the Fetch API and the async/await syntax.

useEffect(() => {

const fetchData = async () => {

const response = await fetch(`https://jsonplaceholder.typicode.com/posts?userId=${userId}`);

const data = await response.json();

setPosts(data);

};

fetchData();

}, [userId]);

Render the list of posts: We use the map function to render the list of posts in the component.

return (

<div>

{posts.map((post) => (

<div key={post.id}>

<h3>{post.title}</h3>

<p>{post.body}</p>

</div>

))}

</div>

);

1. Countdown Timer

Create a CountdownTimer component that displays a countdown timer starting from a given initial value, and stops at 0.

**Expectations**

* Display the countdown timer, starting from the given initial value.
* Update the timer every second using useEffect
* Stop the timer when it reaches 0.
* Display the following text as the timer counts down: "Time Remaining: X"
* Start the timer when the component mounts.
* Stop the timer when the component unmounts.

**Hint:**

Initialize state variable timeRemaining with the initialValue prop: We create a state variable timeRemaining using the useState hook and set its initial value to the initialValue prop.

Set up the useEffect hook to update the timer every second: We use the useEffect hook to create an interval that updates the timeRemaining state every second.

Before setting the interval, we check if timeRemaining is less than or equal to 0. If it is, we don't set the interval, effectively stopping the timer. We also return a cleanup function to clear the interval when the component unmounts or when timeRemaining changes. This ensures that there are no memory leaks or unnecessary intervals running.

useEffect(() => {

if (timeRemaining <= 0) {

return;

}

const timerId = setInterval(() => {

setTimeRemaining((prevTime) => prevTime - 1);

}, 1000);

return () => {

clearInterval(timerId);

};

}, [timeRemaining]);

1. Window Resize Listener

Create a WindowSize component that listens to the window's resize event and displays the current window size. Use the useEffect hook to add and remove the event listener on mount and unmount, respectively.

**Expectations**

* Listen to the window's resize event.
* Display the current window size.
* Add the event listener when the component mounts.
* Remove the event listener when the component unmounts.

**Hint:**

Initialize state variable windowSize with the current window size: We create a state variable windowSize using the useState hook and set its initial value to the current window size.

const [windowSize, setWindowSize] = useState({ width: window.innerWidth, height: window.innerHeight });

Create a handleResize function and add the event listener: We use the useEffect hook to set up the event listener when the component mounts. Inside useEffect, we create a handleResize function that updates the windowSize state with the current window size. We then add the event listener to the window's resize event.

useEffect(() => {

const handleResize = () => {

setWindowSize({ width: window.innerWidth, height: window.innerHeight });

};

window.addEventListener('resize', handleResize);

//...

}, []);

Return a cleanup function to remove the event listener: To ensure proper cleanup, we return a function that removes the event listener using window.removeEventListener. This cleanup function will be called when the component unmounts or if the effect is run again.

useEffect(() => {

//...

return () => {

window.removeEventListener('resize', handleResize);

};

}, []);

Display the current window size: We use the windowSize state variable to display the current window size in a paragraph element.

<p>Window size: {windowSize.width} x {windowSize.height}</p>

1. Form Input Validation

Create a ValidatedInput component that validates user input and shows an error message if the input is invalid. Use the useEffect hook to perform validation whenever the input value changes, simulating componentDidUpdate behavior. This exercise will help you understand how to use the useEffect hook for performing validation based on changes in state and the importance of the dependency array.

**Expectations**

* Validate user input based on a provided validation function.
* Display an error message if the input is invalid.
* Perform validation whenever the input value changes.

**Hint:**

Initialize state variables value and isValid: We create a state variable value using the useState hook and set its initial value to an empty string. We also create a state variable isValid to keep track of the input's validity.

const [value, setValue] = useState('');

const [isValid, setIsValid] = useState(true);

Validate input value with useEffect: We use the useEffect hook to perform validation whenever the input value changes. Inside useEffect, we call the provided validationFunction with the current input value and update the isValid state accordingly. We include value and validationFunction in the dependency array to ensure the effect runs whenever either of them changes.

useEffect(() => {

setIsValid(validationFunction(value));

}, [value, validationFunction]);

Create the input field and update the value state on change: We create an input field and update the value state with the input's value whenever it changes using the onChange event handler.

<input

type="text"

value={value}

onChange={(e) => setValue(e.target.value)}

className={isValid ? '' : 'error'}

/>

Display the error message if the input is invalid: We conditionally render the error message if the isValid state is false.

{!isValid && <p className="error-message">{errorMessage}</p>}

Conclusion

After learning about the useEffect hook in React, you should now have a good understanding of how to handle side effects in functional components.