Hema Coding School

YouTube Link: https://www.youtube.com/@HemaCodingSchool

Introduction

The useEffect hook in React is used to perform side effects in functional components. Side effects can include data fetching, subscriptions, manually changing the DOM, and other operations that run after rendering. It combines the capabilities of various lifecycle methods found in class components (componentDidMount, componentDidUpdate, and componentWillUnmount) into a single API.

```
useEffect(() => {
  // Effect logic here

  return () => {
    // Cleanup logic here
  };
}, [dependencies]);
```

Effect Logic: The main body of the useEffect hook where side effects are performed.
Cleanup Logic: A cleanup function that runs before the component unmounts or before the effect re-runs due to dependency changes.
Dependencies: An array of values that the effect depends on. The effect runs whenever one of these values changes.

Key Concepts

1. **Basic Usage**:
   - Run an effect on every render.

```
useEffect(() => {
  console.log('Component rendered');
});
```

Mounting and Unmounting:

- Run an effect only once after the initial render (mount) and clean up on unmount.

```
useEffect(() => {
  console.log('Component mounted');
  return () => {
    console.log('Component unmounted');
  };
}, []); // Empty dependency array ensures this runs only once
```

Updating:

- Run an effect only when specific dependencies change.

```
useEffect(() => {
  console.log('Count changed:', count);
}, [count]); // Effect runs when `count` changes
```

Combined Usage:

- Handle both mounting, updating, and cleanup in one effect.

```
useEffect(() => {
  // Perform side effect
  console.log('Effect run');

  // Cleanup
  return () => {
    console.log('Cleanup on dependency change or unmount');
  };
}, [dependency]); // Runs on mount and when `dependency` changes
```

Lifecycle methods

Introduction

In React, lifecycle methods are special methods that get called at different stages of a component's life. These stages include mounting, updating, and unmounting. While lifecycle methods are traditionally associated with class components, functional components can achieve similar behavior using hooks, especially the useEffect hook.

Lifecycle Phases

1. **Mounting**: When a component is being inserted into the DOM.
2. **Updating**: When a component is being re-rendered due to changes in props or state.
3. **Unmounting**: When a component is being removed from the DOM.

Hooks for Lifecycle Methods

1. **Mounting**:
   - In class components: componentDidMount.
   - In functional components: useEffect with an empty dependency array [].
2. **Updating**:
   - In class components: componentDidUpdate.
   - In functional components: useEffect with dependencies specified.
3. **Unmounting**:
   - In class components: componentWillUnmount.
   - In functional components: useEffect with a cleanup function.

Using useEffect Hook

The useEffect hook serves as a combination of lifecycle methods in functional components. By customizing the dependencies array, you can control when the side effects run.

```
useEffect(() => {
  // Effect logic here

  return () => {
    // Cleanup logic here
  };
}, [dependencies]);
```

Examples

1. **Component Mounting**:
   - **Purpose**: Run code once after the component mounts.

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

function Component() {
  useEffect(() => {
    console.log('Component mounted');

    // Optional cleanup
    return () => {
      console.log('Cleanup on unmount');
    };
  }, []); // Empty dependency array ensures this runs only once

  return <div>Component Content</div>;
}
```

Component Updating:

- **Purpose**: Run code after the component updates due to changes in dependencies.

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

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

  useEffect(() => {
    console.log('Component updated');
  }, [count]); // Runs whenever `count` changes

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

Component Unmounting:

- **Purpose**: Run cleanup code when the component unmounts

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

function Component() {
  useEffect(() => {
    // Setup code

    return () => {
      console.log('Cleanup on unmount');
    };
  }, []); // Empty dependency array ensures cleanup runs only on unmount

  return <div>Component Content</div>;
}
```

Combining Mounting and Updating:

- **Purpose**: Run code on mount and whenever dependencies change.

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

function Component() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data));

    return () => {
      console.log('Cleanup on update and unmount');
    };
  }, [data]); // Runs on mount and whenever `data` changes

  return (
    <div>
      <p>Data: {data}</p>
    </div>
  );
}
```

Interview Questions:

useEffect in Functional Components

1. **What is the purpose of the useEffect hook in React?**
   - **Answer:** The useEffect hook allows you to perform side effects in functional components. It combines the functionalities of various lifecycle methods from class components (componentDidMount, componentDidUpdate, and componentWillUnmount) into a single API.
2. **How does the dependency array in useEffect work?**
   - **Answer:** The dependency array in useEffect determines when the effect should run. The effect runs after every render if no dependencies are provided. If an empty array is provided, the effect runs only once (after the initial render). If specific dependencies are provided, the effect runs only when one of those dependencies changes.
3. **How do you perform cleanup in useEffect?**
   - **Answer:** You can return a cleanup function from the useEffect hook. This function runs before the component unmounts or before the effect re-runs due to a change in dependencies.

```
useEffect(() => {
  // Effect logic

  return () => {
    // Cleanup logic
  };
}, [dependencies]);
```

How would you mimic componentDidMount using useEffect?

- **Answer:** To mimic componentDidMount, you can use useEffect with an empty dependency array. This ensures the effect runs only once after the initial render.

```
useEffect(() => {
  // Code to run on mount
}, []);
```

How would you mimic componentWillUnmount using useEffect?

- **Answer:** To mimic componentWillUnmount, you return a cleanup function from the useEffect hook. This function runs when the component is about to unmount.

```
useEffect(() => {
  // Setup logic

  return () => {
    // Cleanup logic
  };
}, []); // Empty dependency array ensures this runs only once
```

Explain how you can use useEffect to handle updates to a component.

- **Answer:** You can use useEffect with specific dependencies to handle updates. The effect runs whenever one of the dependencies changes.

```
useEffect(() => {
  // Code to run on update of `dependency`

}, [dependency]); // Effect runs when `dependency` changes
```

Write a useEffect example to fetch data when the component mounts and clean up when it unmounts.

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

function DataFetcher() {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch('https://api.example.com/data');
      const result = await response.json();
      setData(result);
    };

    fetchData();

    return () => {
      // Cleanup if necessary
    };
  }, []); // Empty dependency array ensures this runs only once

  return <div>{data ? <div>{data.name}</div> : <div>Loading...</div>}</div>;
}

export default DataFetcher;
```

Lifecycle Methods in Class Components

1. **What are the main phases of a React component's lifecycle?**
   - **Answer:** The main phases of a React component's lifecycle are:
     - **Mounting**: When the component is being inserted into the DOM. Methods: constructor(), static getDerivedStateFromProps(), render(), componentDidMount().
     - **Updating**: When the component is being re-rendered due to changes in props or state. Methods: static getDerivedStateFromProps(), shouldComponentUpdate(), render(), getSnapshotBeforeUpdate(), componentDidUpdate().
     - **Unmounting**: When the component is being removed from the DOM. Methods: componentWillUnmount().
2. **What is componentDidMount and when is it used?**
   - **Answer:** componentDidMount is a lifecycle method that is invoked immediately after a component is mounted (inserted into the tree). It is often used for initializing network requests, setting up subscriptions, or interacting with the DOM.
3. **What is shouldComponentUpdate and why would you use it?**
   - **Answer:** shouldComponentUpdate is a lifecycle method that determines if a component should re-render based on changes in props or state. By default, it returns true, but you can override it to return false to prevent unnecessary renders, which can improve performance.
4. **How do getSnapshotBeforeUpdate and componentDidUpdate work together?**
   - **Answer:** getSnapshotBeforeUpdate is called right before the changes from the virtual DOM are to be reflected in the DOM. It can capture some information (e.g., scroll position) before the DOM updates. The value returned by getSnapshotBeforeUpdate is passed to componentDidUpdate, where you can use it to handle post-update logic.
5. **What is componentWillUnmount used for?**
   - **Answer:** componentWillUnmount is a lifecycle method that is called just before a component is unmounted and destroyed. It is used to perform cleanup tasks, such as invalidating timers, canceling network requests, or cleaning up subscriptions.