LESSON 06
React useEffect & Component Lifecycle

WEEK 02

Introduction to Component Lifecy

- **What is Lifecycle?**
  Phases a component goes through: Mounting, Updating, Unmounting.
- **Why Important?**
  Manage side effects, optimize performance.
- **React's Approach**
  Hooks (functional) replace class-based lifecycle methods.

Overview of useEffect Hook

- **Definition**
  Hook to handle side effects in functional components.
- **Use Cases**
  Data fetching, subscriptions, DOM updates.
- **Syntax**
  useEffect(() => {}, [dependencies]).

Mounting Phase

- **Definition**
  Component is created and inserted into DOM.
- **Class Equivalent**
  componentDidMount.
- **useEffect Usage**
  Empty dependency array ([]) for mount-only effects.

```
1  export default function MountingPhase() {
2    useEffect(() => {
3      console.log('Effect ran');
4    }, []);
5
6    return <div>Counter</div>;
7  }
```

Updating Phase

- **Definition**
  Component re-renders due to state/prop changes.
- **Class Equivalent**
  componentDidUpdate.
- **useEffect Usage**
  Dependency array with specific values.

```
1  function UpdatingPhase({ count }) {
2    useEffect(() => {
3      console.log(`Count updated: ${count}`);
4    }, [count]);
5
6    return <div>{count}</div>;
7  }
```

Unmounting Phase

- **Definition**
  Component is removed from DOM.
- **Class Equivalent**
  componentWillUnmount.
- **useEffect Usage**
  Return cleanup function from useEffect.

```
1  export default function UnmountingPhase() {
2    useEffect(() => {
3      const timer = setInterval(() => {
4        console.log('Tick');
5      }, 1000);
6
7      return () => {
8        clearInterval(timer);
9      };
10   }, []);
11
12   return <div>Timer</div>;
13 }
```

Dependency Array in useEffect

- **Purpose**
  Controls when effect runs.
- **Options**
  Empty ([]), specific deps, or none (runs every render).
- **Common Mistake**
  Missing dependencies cause bugs.

```
1  export default function DependencyArray({ id }) {
2    useEffect(() => {
3      fetch(`/api/user/${id}`).then((res) => {
4        console.log(res);
5      });
6    }, [id]);
7
8    return <div>User</div>;
9  }
```

Cleanup in useEffect

- return function inside useEffect.
- Runs on unmount or before effect re-runs.
- Good for unsubscribing or removing listeners.
- Example: clearInterval, removeEventListener.
- Prevents memory leaks.
- Similar to componentWillUnmount.

Cleanup in useEffect (Examples)

```tsx
export default function Cleanup() {
  useEffect(() => {
    const timer = setInterval(() => {
      console.log('Tick');
    }, 1000);

    return () => {
      clearInterval(timer);
    };
  }, []);

  return <div>Timer</div>;
}
```

```tsx
function Chat() {
  useEffect(() => {
    const socket = connect();

    return () => {
      socket.disconnect();
    };
  }, []);

  return <div>Chat</div>;
}
```

Fetching Data with useEffect

- **Common Use Case**
  Fetch API data on mount or update.
- **Best Practice**
  Handle loading, error states.

```
1  export default function FetchingData() {
2    const [posts, setPosts] = useState([]);
3    useEffect(() => {
4      fetch('https://jsonplaceholder.typicode.com/posts')
5        .then((res) => res.json())
6        .then((data) => setPosts(data));
7    }, []);
8
9    return (
10     <div>
11       {posts.map((p) => (
12         <p>{p.title}</p>
13       ))}
14     </div>
15   );
16 }
```

Conditional Effects

- **What is it?**
  Run effects based on conditions inside useEffect.
- **Use Case**
  Skip unnecessary API calls.

```
1  function User({ id }) {
2    useEffect(() ⇒ {
3      fetch(`/api/user/${id}`).then((res) ⇒ {
4        console.log(res);
5      });
6    }, [id]);
7
8    return <div>User</div>;
9  }
```

Multiple useEffect Hooks

- **Why Use Multiple?**
  Separate concerns for clarity.
- **Order of Execution**
  Effects run in order of declaration.

```
1  export default function Multiple() {
2    useEffect(() ⇒ {
3      console.log('Effect 1');
4    }, []);
5    useEffect(() ⇒ {
6      console.log('Effect 2');
7    }, []);
8
9    return <div>Multiple</div>;
10 }
```

useEffect vs useLayoutEffect

- **useEffect**
  Runs after render, async.
- **useLayoutEffect**
  Runs before browser paints, sync.
- **Use Case**
  useLayoutEffect for DOM measurements.

```
1  export default function LayoutEffect() {
2    useLayoutEffect(() ⇒ {
3      console.log(document.body.offsetWidth);
4    }, []);
5
6    return <div>Measure</div>;
7  }
```

Comparing Lifecycle Methods vs useEffect

- componentDidMount = useEffect(() => {}, [])
- componentDidUpdate = useEffect(() => {}, [value])
- componentWillUnmount = return () => {}
- Hooks unify logic in one place.
- Class lifecycle is split across methods.
- Hooks are more concise.

Handling Async Operations & Error Handling

❖ Handling Async Operations

- ➢ **Challenge**
  useEffect can't be async directly.
- ➢ **Solution**
  Define async function inside useEffect.

❖ Error Handling

- ➢ **Why Needed?**
  Catch errors in async operations.
- ➢ **Approach**
  Use try-catch inside effect.

```
1  export default function HandlingAsyncOperations() {
2    useEffect(() => {
3      const fetchData = async () => {
4        try {
5          const response = await fetch('https://api.example.com/data');
6          const result = await response.json();
7          console.log('Data fetched successfully:', result);
8        } catch (error) {
9          console.error('Error fetching data:', error);
10       }
11     };
12
13     fetchData();
14   }, []);
15
16   return <div>HandlingAsyncOperations</div>;
17 }
```

Custom Hooks with useEffect

- **Why Custom Hooks?**
  Reuse effect logic across components.
- **Structure**
  Encapsulate useEffect in a hook.

```
1  function useFetch(url) {
2    const [data, setData] = useState(null);
3    useEffect(() => {
4      fetch(url)
5        .then((res) => res.json())
6        .then(setData);
7    }, [url]);
8    return data;
9  }
10
11 export default function CustomHooks() {
12   const data = useFetch('https://api.example.com/data');
13   return <div>{data ? data : 'Loading'}</div>;
14 }
15
```

useEffect with Event Listeners

- **Use Case**
  Add/remove event listeners (e.g., window resize).
- **Cleanup**
  Remove listener in cleanup function.

```
1  export default function EventListeners() {
2    const [size, setSize] = useState(window.innerWidth);
3
4    useEffect(() => {
5      const handleResize = () => {
6        setSize(window.innerWidth);
7      };
8
9      window.addEventListener('resize', handleResize);
10     return () => window.removeEventListener('resize', handleResize);
11   }, []);
12
13   return <div>Width: {size}</div>;
14 }
```

Avoiding Common Pitfalls

- **Missing Dependencies**
  Causes stale data or infinite loops.
- **Overusing Effects**
  Move logic outside useEffect if possible.

```
1  import React, { useState, useEffect } from 'react';
2
3  export default function Pitfalls() {
4    const [count, setCount] = useState(0);
5
6    useEffect(() => {
7      setCount(count + 1); // Infinite loop
8    }, [count]);
9
10   return <div>{count}</div>;
11 }
12
```

Advanced: Chaining Effects

- **What is it?**
  One effect triggers another via state changes.
- **Use Case**
  Sequential API calls.

```
export default function ChainingEffects({ userId }) {
  const [user, setUser] = useState(null);
  const [posts, setPosts] = useState([]);
  useEffect(() => {
    fetch(`/api/user/${userId}`)
      .then((res) => res.json())
      .then(setUser);
  }, [userId]);

  useEffect(() => {
    if (user)
      fetch(`/api/posts/${user.id}`)
        .then((res) => res.json())
        .then(setPosts);
  }, [user]);

  return <div>{posts.length}</div>;
}
```