12. Component Lifecycle, JSX, and Virtual DOM

What

Understand how React handles the lifecycle of components, JSX syntax, and the concept of the virtual DOM. These are core ideas that make React efficient and developer-friendly.

Why

- Lifecycle methods let you control what happens at different stages of a component's life (e.g., when it mounts or updates).
- JSX allows you to write HTML-like syntax directly in JavaScript, making UI development more intuitive.
- The virtual DOM optimizes updates by reducing unnecessary changes in the actual DOM.

How

1. Lifecycle Methods in Functional Components (via Hooks)

The most common lifecycle hooks are:

useEffect: Executes code during mounting, updating, or unmounting phases.

useState: Manages local state.

```
import React, { useState, useEffect } from 'react';
function ExampleComponent() {
  const [count, setCount] = useState(0);
  // Runs on mount and updates
  useEffect(() => {
    console.log(`Count updated: ${count}`);
  }, [count]);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}
export default ExampleComponent;
```

2.JSX Syntax

JSX looks like HTML but is written within JavaScript.

```
const element = <h1>Hello, React!</h1>;
```

- JSX compiles to JavaScript:

```
const element = React.createElement('h1', null, 'Hello, React!');
```

3.Virtual DOM

a. React compares the virtual DOM with the real DOM and updates only the changed parts.

b. Example:

i. Initial Virtual DOM: Hello

ii. New Virtual DOM: Hello, World!

iii. Only the text changes in the real DOM.

---

13. Advanced Hooks (useCallback, useMemo, React.memo)

What

Learn about advanced hooks and techniques to optimize performance in React by avoiding unnecessary re-renders.

Why

Optimization ensures smooth and fast applications, especially for large-scale apps with many components.

How

1. **useCallback** : Prevents function re-creation on every render.

```
import React, { useState, useCallback } from 'react';
const Child = React.memo(({ onClick }) => {
  console.log('Child rendered');
  return <button onClick={onClick}>Click Me</button>;
});

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

  const handleClick = useCallback(() => {
    console.log('Button clicked');
  }, []);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase Count</button>
      <Child onClick={handleClick} />
    </div>
  );
}
export default Parent;
```

2.useMemo: Caches computed values for optimization.

```
import React, { useState, useMemo } from 'react';
function ExpensiveCalculation(num) {
  console.log('Calculating...');
  return num * 2;
}
function App() {
  const [number, setNumber] = useState(1);
  const [toggle, setToggle] = useState(false);
  const double = useMemo(() => ExpensiveCalculation(number), [number]);
  return (
    <div>
      <p>Double: {double}</p>
      <button onClick={() => setNumber(number + 1)}>Increase</button>
      <button onClick={() => setToggle(!toggle)}>Toggle</button>
    </div>
  );
}

export default App;
```

3. React.memo : Prevents unnecessary re-renders for functional components.

```
const MemoizedComponent = React.memo(({ value }) => {
  console.log('Component rendered');
  return <p>{value}</p>;
});
```

---

14. useReducer Hook

What

The useReducer hook is for managing complex state transitions in React.

Why

While useState is simple, useReducer provides a structured approach for managing multiple states or complicated logic.

How

1. **Basic Example**

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

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </div>
  );
}

export default Counter;
```

15. Custom Hooks

What

Custom hooks are functions that allow you to reuse stateful logic across components.

Why

They enable cleaner and more maintainable code by separating reusable logic from components.

How

1. **Creating a Custom Hook**

```
import { useState, useEffect } from 'react';
function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch(url)
      .then((response) => response.json())
      .then((data) => {
        setData(data);
        setLoading(false);
      });
  }, [url]);

  return { data, loading };
}
export default useFetch;
```

2.Using the Custom Hook

```
import useFetch from './useFetch';
function App() {
  const { data, loading } = useFetch('https://jsonplaceholder.typicode.com/posts');
  if (loading) return <p>Loading...</p>;
  return (
    <div>
      {data.map((post) => (
        <h3 key={post.id}>{post.title}</h3>
      ))}
    </div>
  );
}
export default App;
```

---

16. Custom Components and Feature Components

What

Learn to break your app into smaller, reusable components with specific features.

Why

Encourages modularity and reusability, making apps easier to scale and maintain.

How

1. **Building a Feature Component**
    - Example: A reusable card component.

```
function Card({ title, description }) {
  return (
    <div className="card">
      <h3>{title}</h3>
      <p>{description}</p>
    </div>
  );
}
export default Card;
```

2.Using the Card Component

```
import Card from './Card';
function App() {
  return (
    <div>
      <Card title="Card 1" description="This is the first card" />
      <Card title="Card 2" description="This is the second card" />
    </div>
  );
}

export default App;
```