**1. Automatic Batching**

React 18 automatically batches multiple state updates, even if they are inside asynchronous functions, to reduce re-renders.

import React, { useState } from "react";

function App() {

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

const [text, setText] = useState("");

const handleClick = () => {

setCount(count + 1);

setText("Updated");

// In React 17, this would trigger two re-renders.

// In React 18, these are batched into one.

};

return (

<div>

<p>Count: {count}</p>

<p>Text: {text}</p>

<button onClick={handleClick}>Update</button>

</div>

);

}

export default App;

**2. Transition Updates(concurrent rendering)**

Transition updates help differentiate between urgent and non-urgent UI updates. React prioritizes rendering accordingly.

**Transition Updates** in React allow developers to mark certain state updates as **non-urgent**. React prioritizes **urgent updates** (like user input) over these **transitions**, ensuring a smooth user experience, especially for apps with heavy rendering or computations.

React introduced this feature in **React 18**, powered by its **Concurrent Rendering** capabilities.

import React, { useState, useTransition } from "react";

function App() {

const [input, setInput] = useState("");

const [list, setList] = useState([]);

const [isPending, startTransition] = useTransition(); // useTransition hook

const handleChange = (e) => {

const value = e.target.value;

setInput(value);

// Mark list update as non-urgent

startTransition(() => {

const newList = Array(20000)

.fill(null)

.map((\_, index) => `${value} - Item ${index}`);

setList(newList);

});

};

return (

<div>

<h1>React Transition Updates</h1>

<input

type="text"

value={input}

onChange={handleChange}

placeholder="Type something..."

/>

{isPending && <p>Loading...</p>} {/\* Show pending state \*/}

<ul>

{list.map((item, index) => (

<li key={index}>{item}</li>

))}

</ul>

</div>

);

}

export default App;

**How It Works**

1. useTransition**Hook**:
   * React provides the useTransition hook, which returns:
     + A boolean (isPending) to indicate if a transition is in progress.
     + A function (startTransition) to wrap non-urgent state updates.
2. **Urgent vs Non-Urgent Updates**:
   * **Urgent**: Updates like setInput (updates immediately to ensure the input is responsive).
   * **Non-Urgent**: The startTransition wraps the list update, allowing React to prioritize the input over the list rendering.
3. **Pending State**:
   * During the transition, isPending is true, allowing you to display a loader or status message.

**Output**

1. Initially, the app shows an empty input and no list.
2. As you type in the input:
   * The input updates immediately (urgent update).
   * The list rendering starts but shows "Loading..." during the transition.
3. Once the list is rendered, the "Loading..." disappears, and the new list appears.

**3. Suspense for Data Fetching**

React 18 extends Suspense to support asynchronous operations like data fetching.

Suspense for Data Fetching in React is an enhancement to the Suspense component, allowing it to handle asynchronous operations like data fetching. It simplifies loading states by letting React "pause" rendering while data is being fetched and automatically showing a fallback UI (like a loader) until the data is ready.

This feature makes it easier to manage complex asynchronous flows by deferring rendering until all necessary data has been loaded.

import React, { Suspense } from "react";

function App() {

return (

<div>

<h2>User Profile</h2>

<Suspense fallback={<p>Loading user data...</p>}>

<User />

</Suspense>

</div>

);

}

**4. Concurrent Rendering**

Concurrent rendering allows React to interrupt long tasks to keep the UI responsive, such as pausing heavy computations to handle user input.

**Example:**

React doesn't require explicit APIs to enable concurrent rendering. Features like startTransition implicitly use this capability.

**5. Strict Mode Enhancements**

React 18's Strict Mode performs additional checks like mounting and unmounting components twice to ensure side-effects are resilient.

**6. New Root API**

React 18 introduces a new root API for managing the app's lifecycle.

**7. useId Hook**

useId helps generate unique IDs for accessibility and server-side rendering.

**8. Improved Server-Side Rendering (SSR)**

React 18 includes new streaming SSR APIs like renderToPipeableStream and renderToReadableStream for better performance.

**9. useSyncExternalStore**

Provides a way to read and subscribe to external stores, ensuring consistent rendering.

**10. useDeferredValue Hook**

useDeferredValue allows you to defer updates to non-urgent values, improving rendering performance for complex UI.

**11.Batch Processing** is the process of grouping multiple state updates into a single render pass. This minimizes re-renders and improves performance. React has supported **basic batching** for synchronous updates since early versions, and in **React 18**, **automatic batching** extends this capability to asynchronous updates (e.g., promises, setTimeout).

**Key Points:**

* Reduces unnecessary renders by combining multiple state updates.
* Works for both synchronous and asynchronous state updates in React 18.

import React, { useState } from "react";

function App() {

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

const [text, setText] = useState("");

const handleClick = () => {

setTimeout(() => {

setCount((prev) => prev + 1);

setText("Updated!");

// React batches these updates into a single render.

}, 1000);

};

return (

<div>

<p>Count: {count}</p>

<p>Text: {text}</p>

<button onClick={handleClick}>Update</button>

</div>

);

}

export default App;