**🪞 1️⃣ useLayoutEffect — “runs *before* the screen paints”**

**🧠 Think like this:**

* useEffect → runs **after** the browser paints (you *see* the UI first, then it runs)
* useLayoutEffect → runs **before** the browser paints (you don’t see anything yet — perfect for measuring or positioning elements)

**🧩 Example:**

Imagine you want to measure the height or width of a box **right after React places it in the DOM** —  
but before it’s shown to the user (to avoid flicker).

import { useLayoutEffect, useRef, useState } from "react";

function Box() {

const boxRef = useRef();

const [height, setHeight] = useState(0);

useLayoutEffect(() => {

// Runs before paint → you can safely measure

const h = boxRef.current.getBoundingClientRect().height;

setHeight(h);

}, []);

return (

<div>

<div ref={boxRef} style={{ height: "100px", background: "skyblue" }}>

I'm a box

</div>

<p>Height: {height}px</p>

</div>

);

}

🟢 **Why useLayoutEffect?**  
If you used useEffect here, you might see a small “jump” because React would render first → then measure → then update → causing flicker.

**💬 When to use:**

✅ When you need to:

* Measure DOM size or position before painting
* Scroll to a certain position before the user sees anything
* Sync layout changes immediately

⚠️ Not for data fetching or API calls — that’s useEffect.

**🧭 2️⃣ useDebugValue — “for custom hooks (developer only)”**

You rarely use it directly — it’s just to **label custom hook values** in React DevTools.

**🧩 Example:**

import { useState, useDebugValue } from "react";

function useOnlineStatus() {

const [isOnline, setIsOnline] = useState(true);

useDebugValue(isOnline ? "🟢 Online" : "🔴 Offline");

return isOnline;

}

In React DevTools, it’ll now show:

useOnlineStatus: 🟢 Online

✅ Good for debugging, **not for UI logic**.

**⚙️ React 18+ Performance Hooks**

**🚦 3️⃣ useTransition — “make updates non-blocking (for slow UI updates)”**

Let’s say typing in a search box triggers heavy filtering —  
you don’t want the UI to freeze.

import { useState, useTransition } from "react";

function SearchBox({ products }) {

const [query, setQuery] = useState("");

const [filtered, setFiltered] = useState(products);

const [isPending, startTransition] = useTransition();

const handleChange = (e) => {

const value = e.target.value;

setQuery(value);

startTransition(() => {

// This runs in background (non-blocking)

setFiltered(

products.filter((p) => p.toLowerCase().includes(value.toLowerCase()))

);

});

};

return (

<div>

<input value={query} onChange={handleChange} placeholder="Search..." />

{isPending && <p>Loading...</p>}

<ul>

{filtered.map((p, i) => (

<li key={i}>{p}</li>

))}

</ul>

</div>

);

}

✅ UI stays smooth while heavy work happens in background.

**⏳ 4️⃣ useDeferredValue — “delay updating something heavy”**

It’s like saying: “Don’t block the fast parts; update the slow part later.”

import { useState, useDeferredValue } from "react";

function SearchList({ products }) {

const [query, setQuery] = useState("");

const deferredQuery = useDeferredValue(query);

const filtered = products.filter((p) =>

p.toLowerCase().includes(deferredQuery.toLowerCase())

);

return (

<div>

<input value={query} onChange={(e) => setQuery(e.target.value)} />

<ul>

{filtered.map((p, i) => (

<li key={i}>{p}</li>

))}

</ul>

</div>

);

}

✅ UI updates instantly (input stays fast), while results appear a moment later.

**🆔 5️⃣ useId — “unique ID for accessibility or forms”**

Used for generating unique IDs — no conflicts even with SSR.

import { useId } from "react";

function FormField() {

const id = useId(); // gives unique id like "r:abc-1"

return (

<div>

<label htmlFor={id}>Name:</label>

<input id={id} type="text" />

</div>

);

}

✅ Used for accessibility (label–input links), modals, tooltips, etc.  
⚠️ Don’t use for keys in lists — use actual data IDs for that.

**🧩 Summary Table**

| **Hook** | **Purpose** | **When to Use** |
| --- | --- | --- |
| useLayoutEffect | Runs before paint | Measure DOM, avoid flicker |
| useDebugValue | DevTools label | Custom hooks only |
| useTransition | Background updates | Smooth UI during heavy work |
| useDeferredValue | Delay slow updates | Search/filter optimization |
| useId | Unique IDs | Accessibility, form linking |

Would you like me to show one **combined mini example** using useTransition, useDeferredValue, and useId — in a simple product search + form app so you can visually understand their difference?

**You said:**

what is flickering

**ChatGPT said:**

Excellent question 😍 — and super important to understand before moving forward!

Let’s explain **“flickering”** like you’re seeing it live 👀

**💡 What is *flickering*?**

**Flickering** means the UI **visually “jumps,” flashes, or re-renders** too quickly — the user briefly sees something incorrect or unfinished before the final content appears.

It happens when the browser **paints something on the screen**,  
then **immediately changes it again** right after React updates it.

So, you literally *see it change or jump* → that’s a *flicker* ⚡

**🎥 Imagine this real-world example**

You open a React page, and a blue box appears.

1️⃣ React renders it (browser paints).  
2️⃣ Then, in useEffect, you measure the box’s height → update state with setHeight.  
3️⃣ React re-renders (because state changed).  
4️⃣ The blue box momentarily *moves or resizes* — you see that “jump”.

That quick jump = **flicker** 👇

🧠 Because the first render (before measuring) was shown to user → then corrected after.

**🧩 Example code that *causes flicker* using useEffect:**

import { useEffect, useRef, useState } from "react";

function Box() {

const boxRef = useRef();

const [height, setHeight] = useState(0);

useEffect(() => {

// This runs AFTER the browser has already painted the box

const h = boxRef.current.getBoundingClientRect().height;

setHeight(h);

}, []);

return (

<div>

<div ref={boxRef} style={{ height: "100px", background: "skyblue" }}>

I'm a box

</div>

<p>Height: {height}px</p>

</div>

);

}

👉 The browser paints the box first (empty height info),  
then React updates it after — causing a **visual flicker** when the text “Height: 100px” appears.

**✅ Fix it using useLayoutEffect (no flicker):**

import { useLayoutEffect, useRef, useState } from "react";

function Box() {

const boxRef = useRef();

const [height, setHeight] = useState(0);

useLayoutEffect(() => {

// Runs BEFORE paint → measure safely

const h = boxRef.current.getBoundingClientRect().height;

setHeight(h);

}, []);

return (

<div>

<div ref={boxRef} style={{ height: "100px", background: "skyblue" }}>

I'm a box

</div>

<p>Height: {height}px</p>

</div>

);

}

Now, React finishes updating height **before** the screen is shown →  
so the user sees the **final stable UI** immediately — *no flicker* 🎯

**🧠 In short**

| **Concept** | **Happens When** | **User Sees** |
| --- | --- | --- |
| useEffect | After paint | UI shows → updates → small jump (flicker) |
| useLayoutEffect | Before paint | UI shows in final state (no flicker) |

**🎨 1️⃣ What “browser paint” means**

When React finishes **rendering the virtual DOM**, the browser takes that HTML and **paints it visually on screen**.  
So the order is:

Render virtual DOM → Commit to real DOM → Paint (user sees)

**🧩 2️⃣ When Hooks run in that timeline**

| **Hook** | **Runs** | **What happens** |
| --- | --- | --- |
| useLayoutEffect | 🕐 **Right after DOM is updated but before paint** | You can read/write layout safely (measure height, scroll, etc.) |
| useEffect | 🕐 **After paint** | You can do async stuff (API calls, logs, subscriptions) |

So, useLayoutEffect runs **after** React updates the DOM but **before the browser shows it** to the user.  
That’s why if you measure or modify DOM in that tiny window — the user never sees the “jump.”

**👁️ 3️⃣ What the user actually sees**

Let’s visualize it with your question 👇

**Example:**

We want to:

* Call API to fetch products
* Then measure the DOM element height

**⚙️ Step-by-step timeline:**

**🧱 Initial Render**

* Screen shows **Loading...**
* DOM = <p>Loading...</p> → Browser paints it.
* useEffect (API call) starts after paint.

So user **sees the loading text** — that’s fine.

**🛰️ API Returns**

* State updates with fetched data → React re-renders.
* New DOM = Product list rendered.

Now:

* Before browser paints this **new updated DOM**, React runs useLayoutEffect.

That’s the key difference.  
➡️ You can now measure offsetHeight or update layout **before** the user sees it.

So visually:

The browser shows the *already measured, correct* version — no flicker or jump.

**⚡️ 4️⃣ What if we used useEffect instead?**

If you measure DOM in useEffect, it runs *after* paint.

So:

* Browser paints product list 🖼️
* Then you measure and set new layout (like height or position)
* Then React re-renders again

This causes a **small visible jump (flicker)** — because the browser first showed the wrong layout and then corrected it.

**💡 Summary**

| **Hook** | **When it runs** | **Visible effect** |
| --- | --- | --- |
| useLayoutEffect | Before paint | No flicker |
| useEffect | After paint | Might flicker (since DOM is already visible) |

So, to your exact line 👇

“It doesn’t show anything on screen only until API is called?”

Not exactly.

* The *Loading state* is already shown (painted).
* After API returns, React re-renders.
* useLayoutEffect runs **before painting the new UI**, so the user sees the corrected, measured layout instantly.

Top of Form

Bottom of Form