React Loading-Strategie mit hasFetchedRef

Eine kurze, praxisnahe Erklärung für Junior-Entwickler:innen

Was ist das Problem?

Bei Seiten oder Cards, die Daten über fetch() laden, sieht man beim allerersten Render oft nur einen leeren Zustand oder sogar die Meldung „Keine Daten vorhanden.“ – und erst beim zweiten Render wechseln wir auf einen echten Loading-State. Das fühlt sich wie Flackern an und ist nicht schön.

Ziel

Beim ersten Aufruf soll sofort ein Skeleton erscheinen – noch bevor der Request überhaupt losläuft – und erst nachdem der erste Request abgeschlossen ist, dürfen wir auf den normalen Loading-/Fehler-/Daten-Zustand umschalten.

Die Idee hinter hasFetchedRef

Wir benutzen ein useRef-Flag (hasFetchedRef), das merkt, ob der erste Daten-Request schon einmal vollständig durchgelaufen ist. Solange das nicht der Fall ist, wird immer ein Skeleton angezeigt. Danach verhält sich die Komponente wie gewohnt (z. B. bei späteren Refreshes oder Tab-Wechseln).

```
const hasFetchedRef = useRef(false);

useEffect(() => {
  const ac = new AbortController();
  (async () => {
    try {
      setIsLoading(true);
      const res = await fetch(API_URL, { signal: ac.signal });
      if (!res.ok) throw new Error(`HTTP ${res.status}`);
      const json = await res.json();
      setData(json);
    } catch (err) {
      if (err?.name !== "AbortError") {
        console.error(err);
        setData(null);
      }
    } finally {
      if (!ac.signal.aborted) {
        setIsLoading(false);
        hasFetchedRef.current = true; // Erster Request abgeschlossen
      }
    }
  })();
  return () => ac.abort();
}, []);

// Anzeige-Logik:
const showSkeleton = !hasFetchedRef.current || isLoading;

return (
  <Card>
    {showSkeleton ? <SkeletonChart /> : data ? <RealChart data={data} /> : <EmptyState />}
  </Card>
);
```

Wieso reicht isLoading alleine nicht?

isLoading wird in vielen Stores oder Komponenten erst im useEffect gesetzt. Beim allerersten Render ist es daher noch false → du würdest kurz den Leerlauf/Empty-State sehen, bevor der Loading-State greift. hasFetchedRef fängt genau diesen Moment ab.

Verhalten in drei Zuständen

- 1. Initial (noch nie geladen): showSkeleton ist true ⇒ Skeleton wird angezeigt.
- 2. Während des ersten Requests: isLoading ist true ⇒ Skeleton bleibt sichtbar.
- 3. Nach dem ersten Request: hasFetchedRef = true ⇒ normales Verhalten (Daten/Fehler/kein Datensatz).

Wiederverwendbare Skeletons für gleichbleibendes Layout

Um Layout-Sprünge zu vermeiden, sollten Skeletons die gleiche Höhe, Margins und Achsenbereiche wie die echten Charts/Listenelemente haben. So sehen Nutzer:innen während des Ladens ein stabiles Layout ohne visuelles Flackern.

```
// Beispiel: SkeletonBarChart nutzt die gleiche Höhe/Margins wie dein echter BarChart
<SkeletonBarChart
  height={205}
  bars={10}
  margin={{ top: 8, right: 12, left: 10, bottom: 8 }}
  barGap={8}
  maxBarHeight={140}
  showXAxisLine
/>
```

Mini-Beispiel (End-to-End)

```
export default function DashboardCardExample() {
  const [isLoading, setIsLoading] = useState(false);
  const [data, setData] = useState<any[]>([]);
  const hasFetchedRef = useRef(false);

  useEffect(() => {
    const ac = new AbortController();
    (async () => {
      try {
        setIsLoading(true);
        const res = await fetch("/api/data", { signal: ac.signal });
        if (!res.ok) throw new Error(`HTTP ${res.status}`);
        const json = await res.json();
        setData(json?.items ?? []);
      } catch (e) {
        if (e?.name !== "AbortError") {
          console.error(e);
          setData([]);
        }
      } finally {
        if (!ac.signal.aborted) {
          setIsLoading(false);
          hasFetchedRef.current = true;
        }
      }
    })();
    return () => ac.abort();
  }, []);

  const showSkeleton = !hasFetchedRef.current || isLoading;

  return (
    <Card className="card" elevation={4}>
      <CardContent>
        {showSkeleton ? <SkeletonBarChart height={205} /> : <RealBarChart data={data} />}
      </CardContent>
    </Card>
  );
}
```

Häufige Stolperfallen

- hasFetchedRef vergessen zurückzusetzen? In einer einzelnen Card i. d. R. nicht nötig. Wenn die Card aber bewusst zurückgesetzt werden soll (z. B. Filterwechsel), kannst du hasFetchedRef.current = false setzen.
- AbortController nicht genutzt? Bei schnellen Navigationswechseln vermeidest du so State-Updates nach Unmount.
- Skeleton-Höhe passt nicht? Achte auf identische Margins/Höhen/Achsen wie im echten Chart. Dann gibt es keinen Layout-Shift.

Wann eignet sich diese Strategie?

• Für Cards/Widgets, die beim ersten Mount Daten laden und ein stabiles, ruhiges UI zeigen sollen. • Wenn der Store/das isLoading-Flag erst im Effect gesetzt wird und du kurzes Flackern vermeiden willst. • Für Skeletons, die genau wie das reale UI aussehen (keine Höhen-Sprünge).

Erweitert: Zusammenspiel mit globalem Store-Loading

Du kannst weiterhin ein globales isLoading im Zustand-Store pflegen (z. B. für Spinners oder globale Loader). hasFetchedRef ist eine rein lokale Optimierung pro Card/Komponente, die nur den allerersten Ladezustand steuert. Danach übernimmt dein gewohntes Pattern.

Vergleich: Nur isLoading vs. hasFetchedRef + isLoading

Variante	Erstes Render	User Experience
Nur isLoading	Kurz leer/Empty, dann Loading	Kleines Flackern möglich
hasFetchedRef + isLoading	Sofort Skeleton	Stabil & ruhig, kein Flackern

Faustregel

Wenn eine Card beim ersten Mount Daten lädt: Nutze hasFetchedRef, um direkt Skeletons zu zeigen. Für alle weiteren Updates (Refetch, Filter, Pagination) reicht dein reguläres isLoading.

Viel Erfolg beim Implementieren und Happy Shipping! ■