011-Skeleton-Dashboard

Frontend-Aufgabe
Anforderungsniveau: Einfach

1 Ziel der Aufgabe

Sie bauen eine Übersichtsseite, die drei Datenbereiche aus einer künstlich verzögerten Datenquelle lädt. Jeder Bereich hat eine andere Antwortzeit, einer kann auf Knopfdruck fehlschlagen, einer liefert eine leere Menge. Geprüft wird, ob Sie Lade-, Fehler- und Leerzustand sauber voneinander trennen und ob Ihre Platzhalter das Layout so stabil halten, dass beim Eintreffen der Daten nichts springt.

Die Aufgabe ist bewusst klein gehalten. Es geht nicht um Datenmengen oder Geschäftslogik, sondern um die Sorgfalt in den Zuständen dazwischen.

Veranschlagte Bearbeitungszeit: 2–3 Stunden.

Verwendete Techniken:

- React mit TypeScript, wahlweise Next.js (App Router) oder Vite
- Datenabruf mit eigenem Ladezustand je Bereich
- Skeleton-Platzhalter in Zielgröße statt globalem Spinner
- Fehlerbehandlung mit Wiederholen-Aktion

2 Aufgabenbeschreibung

1. **Projekt aufsetzen**

   Legen Sie ein neues Projekt an. Beide Varianten sind zulässig:

```
npx create-next-app@latest skeleton-dashboard --typescript --app
# oder
npm create vite@latest skeleton-dashboard -- --template react-ts
```

   Für die Gestaltung ist Ihnen die Wahl freigestellt (CSS-Module, Tailwind CSS, reines CSS). Das Aussehen wird nicht bewertet – Farben, Schriften und Feinschliff sind gleichgültig, schlichtes CSS genügt. Bewertet wird, dass die Zustände unterscheidbar sind und das Layout nicht springt.

2. **Verzögerte Datenquelle bereitstellen**

   Es wird kein Server und kein Endpunkt benötigt. Legen Sie ein Modul mit drei Funktionen an, die feste Beispieldaten nach einer künstlichen Wartezeit als Promise zurückgeben:

```
// src/daten.ts
ladeStammdaten()    ~  300 ms   1 Datensatz mit Logo
ladeAktivitaeten()  ~ 2500 ms   5 Eintraege (Zeitstempel, Benutzer, Vorgang)
ladeMeldungen()     ~  800 ms   leeres Array

const warte = (ms: number) => new Promise((r) => setTimeout(r, ms));
```

```
export async function ladeStammdaten(): Promise<Stammdaten> {
  await warte(300);
  return { kunde: "Beispiel Logistik GmbH", kundennummer: "K-100428",
           anschrift: "Musterweg 3, 01067 Dresden",
           logo: "/platzhalter-logo.svg" };
}
```

Das Logo müssen Sie nicht entwerfen. Legen Sie diese Datei unverändert unter `public/platzhalter-logo.svg` ab:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 120 120"
     width="120" height="120">
  <rect width="120" height="120" rx="12" fill="#c8d2dc"/>
  <text x="60" y="72" font-family="sans-serif" font-size="34"
        text-anchor="middle" fill="#40525f">BL</text>
</svg>
```

Die Wartezeiten dürfen fest verdrahtet sein.

3. **Übersichtsseite bauen**

   Bauen Sie eine Seite mit drei Bereichen, die ihre Daten **unabhängig voneinander** laden. Kein Bereich wartet auf einen anderen, und die Seite zeigt keinen globalen Ladebildschirm.

   - **Kundensteckbrief** – ein Kopfbereich mit dem Logo links und den Stammdaten daneben, je Zeile eine Beschriftung und ihr Wert
   - **Letzte Aktivitäten** – Liste mit Zeitstempel, Benutzer und Vorgang
   - **Systemmeldungen** – Liste, die in den Beispieldaten leer ist

4. **Ladezustand als Skeleton**

   Jeder Bereich zeigt während des Ladens einen Platzhalter, der die Form des späteren Inhalts hat: ein Kasten in Logogröße mit danebenliegenden Textzeilen unterschiedlicher Breite, fünf Zeilen-Platzhalter für die Aktivitäten. Der Platzhalter muss dieselbe Höhe und Breite einnehmen wie der fertige Inhalt und trägt `aria-hidden`.

   Besondere Sorgfalt verlangt das Logo: Der Platz dafür muss vor dem Laden des Bildes feststehen, etwa über feste Maße oder `aspect-ratio`. Ein Bild, das seine Größe erst beim Eintreffen mitbringt, schiebt die Stammdaten daneben zur Seite – genau das soll nicht passieren.

   Die Prüfung ist einfach: Öffnen Sie die Seite, schauen Sie beim Laden hin und nach dem Laden noch einmal. Die Blöcke müssen an derselben Stelle stehen und dieselbe Größe haben.

5. **Fehlerzustand**

   Bauen Sie an den Bereich „Letzte Aktivitäten“ eine Bedienmöglichkeit, mit der sich ein Fehlschlag zuschalten lässt, etwa ein kleiner Schalter. Eine Codeänderung darf dafür nicht nötig sein.

   Schlägt der Abruf fehl, zeigt **nur dieser Bereich** eine Fehlermeldung mit Ursache und einer Schaltfläche „Erneut versuchen“, die genau diesen Abruf wiederholt. Die übrigen Bereiche bleiben unberührt und behalten ihre Daten. Während der Wiederholung erscheint wieder der Skeleton des Bereichs.

6. **Leerzustand**

   Der Bereich „Systemmeldungen“ liefert eine leere Liste. Er muss sich deutlich von Laden und Fehler unterscheiden: eine erklärende Zeile im Sinne von „Keine Systemmeldungen vorhanden“, nicht ein leerer Kasten und nicht ein dauerhaft laufender Skeleton. Ein leeres Ergebnis ist ein erfolgreicher Abruf.

3 Anforderungen

- Die drei Bereiche laden unabhängig voneinander. Ein langsamer Bereich blockiert keinen schnellen.
- Für Inhaltsflächen werden **Skeleton-Platzhalter** verwendet, keine Spinner.
- **Kein Layout-Sprung.** Platzhalter belegen die Zielgröße; das Logo reserviert seinen Platz, bevor das Bild geladen ist.
- Laden, Fehler, Leer und Erfolg sind vier unterscheidbare Zustände. Ein leeres Ergebnis wird nie als Fehler dargestellt.
- Der Fehlerzustand ist auf den betroffenen Bereich begrenzt und wiederholbar.
- TypeScript ohne `any` in den Datentypen.

4 Abgabe

- Git-Repository. Der Ordner `node_modules` gehört nicht hinein.
- `README.md` mit der Startanleitung und einem Hinweis, wie Fehler- und Leerzustand vorgeführt werden.
- Das Projekt startet nach `npm install` und `npm run dev` ohne weitere Handgriffe.

5 Bewertungskriterien

- **Zustandstrennung** – sind Laden, Fehler, Leer und Erfolg wirklich vier Fälle, oder fällt einer in einen anderen hinein?
- **Layoutstabilität** – springt beim Übergang vom Platzhalter zum Inhalt etwas, rutscht der Steckbrief beim Eintreffen des Logos, wächst ein Kasten?
- **Unabhängigkeit der Bereiche** – erscheinen die schnellen Daten sofort, oder wartet die Seite auf den langsamsten Abruf?
- **Wiederholbarkeit** – funktioniert „Erneut versuchen“ ohne Seitenneuladen und ohne die übrigen Bereiche zurückzusetzen?
- **Codequalität** – wiederverwendbare Skeleton-Komponenten statt kopierter Blöcke, sauber typisierte Datenstrukturen, verständliche Benennung