015-Tabelle-Serverseitige-Paginierung

Frontend-Aufgabe
Anforderungsniveau: Mittel

1 Ziel der Aufgabe

Sie bauen eine Datentabelle, die mehrere zehntausend Datensätze anzeigt, ohne sie jemals vollständig zu laden. Seite, Sortierung und Suche gehen an den Server, der genau eine Seite zurückliefert. Geprüft wird, ob Sie den Ansichtszustand sauber in der URL führen, die Ladezustände ohne Layout-Sprung darstellen und mit nebenläufigen Datenabfragen umgehen.

Veranschlagte Bearbeitungszeit: 4–6 Stunden

Abgrenzung zur Aufgabe 009

Unsere Aufgabe 009 („Progressive Product Catalog“) verlangt ebenfalls Liste, Sortierung, Suche und Filter – und sieht auf den ersten Blick ähnlich aus. Sie ist das genaue Gegenteil dieser Aufgabe, und darauf kommt es hier an:

- In 009 liegen *alle* Daten im Browser (lokale JSON-Datei, `localStorage`), und Filtern, Sortieren und Blättern sind Operationen auf einem Array im Speicher. Genau dieser Codepfad ist hier verboten: Es darf keine Stelle geben, die mehr als eine Seite holt und im Browser schneidet.
- In 009 lebt der Ansichtszustand in React-State oder einem Store. Hier ist die **URL die einzige Quelle der Wahrheit**. Ein zweiter, danebenliegender `useState` für Seite oder Sortierung gilt als Fehler, nicht als Geschmacksfrage.
- 009 kennt keine Nebenläufigkeit, weil nichts über das Netz geht. Hier kommen Antworten in beliebiger Reihenfolge zurück, und der Umgang damit ist ein Kernkriterium.

Wenn Sie 009 bereits bearbeitet haben: Übernehmen Sie daraus nichts. Die Lösung von 009 ist hier definitionsgemäß falsch.

2 Aufgabenbeschreibung

1. **Datenquelle bereitstellen**

   Sie brauchen einen Endpunkt, der genug Daten hat, dass ein vollständiges Laden keine Option ist. Erzeugen Sie einmalig einen Datensatz von mindestens 20.000 Bestellungen – etwa mit `@faker-js/faker` (deutsche Namen liefert `fakerDE`) und einem festen Startwert über `seed()`, damit die Daten reproduzierbar sind – und legen Sie ihn als JSON-Datei oder in einer SQLite-Datenbank ab. Erfundene Daten, keine echten Kundennamen oder Kennungen.

   Den Endpunkt setzen Sie als Next.js Route Handler unter `app/api/orders/route.ts` um oder als kleinen eigenen Node-Dienst (etwa mit Express oder Fastify). Fertige Attrappen wie `json-server` sind hier nicht geeignet – deren Parameternamen und Antwortform liegen fest und passen nicht auf den folgenden Vertrag. Der Endpunkt muss diese Parameter verstehen:

```
GET /api/orders?page=3&pageSize=25&sort=total&dir=desc&q=mueller

page      1-basiert, Standard 1
pageSize  10 | 25 | 50, Standard 25
sort      id | customer | createdAt | total, Standard id
dir       asc | desc, Standard asc
q         Volltextsuche über Kundenname und Bestellnummer
```

Die Antwort enthält die Seite und die Gesamtzahl der Treffer nach Suche:

```
{
  "items": [ { "id": "ORD-0000042", "customer": "Anna Beispiel",
               "createdAt": "2025-11-04T09:12:00Z", "total": 1249.9,
               "currency": "EUR" } ],
  "page": 3, "pageSize": 25, "total": 1873
}
```

Ungültige Parameter führen nicht zum Absturz, sondern zum Standardwert oder zu Status 400 mit verständlicher Meldung. `pageSize` ist serverseitig begrenzt – ein Aufruf mit `pageSize=100000` darf die Grenze nicht aushebeln. Bauen Sie zum Testen eine künstliche Verzögerung von 300 bis 800 ms ein, damit die Ladezustände überhaupt sichtbar werden.

2. **Tabelle mit serverseitiger Paginierung**

   Die Ansicht lädt ausschließlich die aktuelle Seite. Es gibt keinen Codepfad, der alle Datensätze holt und im Browser schneidet.

   - Spalten: Bestellnummer, Kunde, Datum, Betrag.
   - Beträge rechtsbündig, formatiert mit `Intl.NumberFormat`; Datumsangaben in deutscher Schreibweise mit `Intl.DateTimeFormat`.
   - Seitennavigation (erste, vorherige, nächste, letzte Seite), eine Anzeige der Spanne („26 bis 50 von 1.873“) und ein Umschalter für die Seitengröße.
   - Sortierung per Klick auf den Spaltenkopf: aufsteigend, absteigend, zurück zur Standardsortierung (`id` aufsteigend) – sichtbar und über `aria-sort` lesbar.
   - Ein Suchfeld für die Volltextsuche.

   Für die Tabellenlogik dürfen Sie `@tanstack/react-table` im Modus `manualPagination` und `manualSorting` verwenden oder die Tabelle von Hand bauen. Begründen Sie die Wahl im README.

3. **Zustand in der URL**

   Seite, Seitengröße, Sortierspalte, Sortierrichtung und Suchbegriff stehen vollständig in der Query-Zeichenkette. Die URL ist damit teilbar: wer sie kopiert und in einem neuen Tab öffnet, sieht exakt dieselbe Ansicht.

   ```
   /orders?page=3&pageSize=25&sort=total&dir=desc&q=mueller
   ```

   Daraus ergeben sich Regeln, die Sie einhalten müssen:

   - Kein doppelter Zustand: Seite und Sortierung kommen aus der URL, nicht aus einem zusätzlich mitgeführten `useState`.
   - Ändert sich die Suche oder die Seitengröße, springt die Seitenzahl auf 1. Sonst landen Nutzer auf Seite 7 eines Ergebnisses mit zwei Seiten.

- Der Zurück-Knopf führt zur vorherigen Ansicht. Überlegen Sie, welche Änderungen einen Verlaufseintrag verdienen (Seite, Sortierung) und welche nicht (jeder Tastendruck im Suchfeld) – `router.push` gegen `router.replace`.
- Eine unsinnige URL ergibt keine kaputte Ansicht: Parameter beim Einlesen prüfen, Unbrauchbares durch den Standardwert ersetzen.

Sie dürfen `useSearchParams` mit `URLSearchParams` direkt nutzen oder eine Bibliothek wie `nuqs` einsetzen.

4. **Nebenläufigkeit und Eingabeverhalten**

   Wer schnell durch die Seiten klickt, löst mehrere Anfragen aus, die in beliebiger Reihenfolge zurückkommen können. Die Tabelle darf nie das Ergebnis einer veralteten Anfrage zeigen.

   - Laufende Anfragen per `AbortController` abbrechen oder Antworten verwerfen, die nicht zur aktuellen Anfrage gehören. Wer `@tanstack/react-query` einsetzt, erklärt im README, welcher Mechanismus dort greift.
   - Die Sucheingabe wird entprellt (etwa 300 ms).
   - Fehlgeschlagene Anfragen erzeugen einen sichtbaren Fehlerzustand mit einem Knopf zum erneuten Versuch, nicht eine leere Tabelle.

5. **Ladezustände ohne Layout-Sprung**

   Während eine Seite nachgeladen wird, zeigt die Tabelle Skeleton-Zeilen in der Anzahl der eingestellten Seitengröße und in der Höhe echter Zeilen. Spaltenbreiten bleiben stabil, Tabellenkopf und Seitennavigation verschwinden nicht – nichts darf verrutschen, wenn die Daten eintreffen. Für kurze Aktionen am Auslöser ist ein Spinner richtig, für die Inhaltsfläche nicht. Ist das Ergebnis leer, erscheint ein eigener Zustand mit Hinweis und einem Knopf „Suche zurücksetzen“.

6. **Tests**

   Zwei Tests genügen, wenn sie das Richtige nachweisen – nämlich, dass wirklich serverseitig paginiert wird, und nicht nur, dass die Tabelle Zeilen rendert.

   - Eine Einheitsprüfung für das Umwandeln zwischen URL-Parametern und Ansichtszustand, inklusive ungültiger Werte (`page=-1`, `sort=drop`).
   - Ein Komponententest mit nachgestellter Netzwerkschicht (etwa `msw`), der belegt: ein Klick auf den Spaltenkopf schickt `sort` und `dir` an den Server, und die Antwort einer überholten Anfrage überschreibt die neuere nicht.

3 Anforderungen

- React mit TypeScript, vorzugsweise Next.js (App Router). Kein `any` in Ihrem Code; die Antwort des Endpunkts ist typisiert.
- Eine Anfrage holt nie mehr als eine Seite; der gesamte Datenbestand wird nie in den Browser geladen. Ein Test oder ein Blick ins Netzwerkprotokoll muss das belegen können.
- Seite, Seitengröße, Sortierung und Suche stehen in der URL und überleben ein Neuladen der Seite. Kein zweiter Zustand daneben.
- Skeleton-Zeilen in Zielgröße für die Tabellenfläche, Spinner nur für kurze Aktionen an ihrem Auslöser. Kein Layout-Sprung beim Seitenwechsel.

- Semantisches Markup: `table`, `thead`, `tbody`, `th` mit `scope`. Sortierbare Köpfe sind Schaltflächen, tastaturbedienbar und mit `aria-sort` ausgezeichnet; Ladevorgänge werden über eine Live-Region angekündigt.
- Das Aussehen darf schlicht bleiben. Bewertet wird das Verhalten, nicht die Gestaltung; investieren Sie die Zeit lieber in Punkt 3 und 4.
- Kein Formatierer ohne Projektkonfiguration im Repository. Wenn Sie Prettier oder ESLint einsetzen, legen Sie die Konfigurationsdatei mit ab.
- Nur tatsächlich vorhandene Bibliotheken, alle in `package.json` eingetragen. Keine echten Kunden- oder Zugangsdaten in den Testdaten.

4 Abgabe

- Ein Git-Repository mit nachvollziehbarer Historie. Keine einzelne Ablage „Initial commit“ mit dem fertigen Ergebnis.
- `README.md` mit Startanleitung, Erzeugung des Testdatensatzes, Testbefehl, einer kurzen Beschreibung, wie der Ansichtszustand zwischen URL, Anfrage und Tabelle fließt, und einer Notiz zu bewusst Weggelassenem.
- Das Projekt startet ohne weitere Handgriffe:

```
npm install
npm run seed     # erzeugt den Testdatensatz
npm run dev
npm test
```

- `.gitignore` für `node_modules`, Build-Verzeichnis und die erzeugte Datendatei.

5 Bewertungskriterien

- **Wirklich serverseitig** – keine Stelle im Code, die mehr als eine Seite lädt; Seitengrenzen, letzte Seite, Gesamtzahl nach Suche, Verhalten bei leerem Ergebnis.
- **URL-Zustand** – teilbar, wiederherstellbar, Zurück-Knopf funktioniert, kein zweiter Zustand daneben.
- **Umgang mit Nebenläufigkeit** – keine veralteten Antworten in der Tabelle, sinnvolle Entprellung.
- **Ladezustände** – Skeleton in Zielgröße, kein Springen des Layouts, verständlicher Fehlerzustand.
- **Zugänglichkeit** – Tabellensemantik, Tastaturbedienung, `aria-sort`, Ankündigung von Aktualisierungen.
- **Tests** – prüfen sie das Verhalten oder nur, dass etwas gerendert wurde? Aussagekräftige Tests zählen mehr als viele.
- **Codequalität und Dokumentation** – Typisierung, Trennung von Datenabruf und Darstellung, lesbare Namen; lässt sich das Projekt ohne Rückfrage starten?