019-Echtzeit-Dashboard

Frontend-Aufgabe
Anforderungsniveau: Schwer

1 Ziel der Aufgabe

Sie bauen ein Dashboard, das Messwerte aus mehreren Quellen laufend entgegennimmt und als Verlauf sowie als Statusübersicht darstellt. Der interessante Teil ist nicht das Zeichnen der Kurven, sondern das Verhalten an den Rändern: Was zeigt die Oberfläche, während die Verbindung wackelt oder weg ist? Wie kommt sie zurück, ohne den Server mit Versuchen zu überrollen? Und wie kommen die während der Trennung entstandenen Werte nachträglich in den Verlauf – lückenlos und ohne Dubletten?

Die Aufgabe ist bewusst auf diesen Kern beschränkt. Filter, Themenwechsel und ähnliches Beiwerk sind nicht gefragt.

Veranschlagte Bearbeitungszeit: 6–8 Stunden

Geprüft werden:

- React mit TypeScript im strict-Modus, sauber typisierte Zustandsmodelle
- Server-Sent Events als Transport, inklusive Wiederaufbau
- Exponentielles Backoff mit Jitter, Obergrenze und Rücksetzung
- Nachladen verpasster Werte über eine Sequenznummer, Zusammenführen mit dem laufenden Strom
- Umgang mit Datenmengen im Frontend: Ringpuffer, gebündeltes Rendern
- Zustandsanzeige, Barrierefreiheit und Tests für die Fälle, die man nicht sieht

2 Aufgabenbeschreibung

1. **Datenquelle bereitstellen**

   Schreiben Sie einen kleinen Mock-Server (Node mit TypeScript, z. B. Express oder Fastify), der sechs Quellen simuliert und je Quelle etwa fünf Werte pro Sekunde erzeugt. Jeder Wert trägt eine je Quelle streng monoton steigende Sequenznummer.

```
{
  "quelle": "sensor-a3",
  "seq": 10431,
  "zeit": "2026-01-14T09:12:03.120Z",
  "wert": 21.4,
  "einheit": "C",
  "status": "ok"
}
```

   Der Server bietet drei Datenendpunkte und einen Steuerbefehl:

```
GET  /api/quellen      Liste aller Quellen: Name, Einheit, Grenzwerte
GET  /api/strom        SSE-Strom, ein Ereignis je Messwert
GET  /api/messwerte?quelle=sensor-a3&seit=10420&limit=500
                       { "werte": [...], "vollstaendig": true, "aeltester": 10120 }
POST /api/steuerung/stoerung   { "dauerSekunden": 20 }
                       # trennt bestehende Ströme und nimmt für die
                       # angegebene Dauer keine neue Verbindung an
```

Der Strom nimmt nichts wieder auf: Nach einem Abbruch beginnt er bei den aktuellen Werten, alles dazwischen fehlt. Die Lücken schließen Sie ausschließlich je Quelle über `/api/messwerte`.

Der Server hält je Quelle nur die letzten 2000 Werte vor. Liegt `seit` davor, antwortet er mit `vollstaendig: false` und dem ältesten noch verfügbaren Wert – dieser Fall muss im Frontend sichtbar werden und darf nicht stillschweigend als lückenloser Verlauf durchgehen. Machen Sie die Vorhaltemenge einstellbar (Umgebungsvariable oder Startparameter), sonst müsste ein Prüfer minutenlang warten, um ihn überhaupt auszulösen.

2. **Verbindung als Zustandsmaschine**

   Modellieren Sie die Verbindung als expliziten, typisierten Zustand – nicht als drei lose Boolean-Flags.

```
type Verbindung =
  | { zustand: "initial" }
  | { zustand: "verbindet"; versuch: number }
  | { zustand: "verbunden"; seit: number }
  | { zustand: "wartet"; versuch: number; naechsterVersuchUm: number }
  | { zustand: "aufgegeben"; versuche: number };
```

   Der Zustand ist jederzeit in der Kopfzeile sichtbar: Zustand, Zeitpunkt des letzten empfangenen Wertes und – im Zustand `wartet` – ein herunterzählender Sekundenwert bis zum nächsten Versuch, dazu eine Schaltfläche „Jetzt verbinden“. Zustandswechsel werden über eine Live-Region (`aria-live` mit dem Wert `polite`) angesagt.

3. **Wiederaufbau mit Backoff**

   Nach einem Abbruch wird erneut verbunden, mit wachsender Wartezeit und Streuung:

```
wartezeit(versuch) = min(basis * 2^(versuch-1), obergrenze)
effektiv           = zufall(0, wartezeit)      // volle Streuung
```

   Basis 1000 ms, Obergrenze 30 s. Der Zähler wird zurückgesetzt, sobald eine Verbindung 10 Sekunden stabil stand – nicht schon beim Verbindungsaufbau, sonst entsteht bei einem Server, der sofort wieder trennt, eine Dauerschleife ohne Backoff. Nach 8 vergeblichen Versuchen geht die Anwendung in `aufgegeben` und verbindet nur noch auf ausdrückliche Aktion weiter. Die Berechnung gehört in eine reine Funktion ohne Timer und ohne React.

4. **Verpasste Werte nachziehen**

   Merken Sie sich je Quelle die zuletzt verarbeitete Sequenznummer. Nach jedem erfolgreichen Wiederaufbau werden die Lücken geschlossen:

   - Für jede Quelle die fehlenden Werte über `/api/messwerte` anfordern, bei Bedarf seitenweise.
   - Während des Nachladens laufen bereits neue Werte über den Strom ein. Diese dürfen nicht verloren gehen und nicht doppelt im Verlauf landen – puffern Sie sie und führen Sie beides über die Sequenznummer zusammen.

- Ist der Abstand größer als das, was der Server noch vorhält, wird die Lücke als solche in den Verlauf eingetragen und in der Kurve als Unterbrechung gezeichnet, nicht durch eine gerade Linie überbrückt.
- Werte, die verspätet oder doppelt eintreffen, werden verworfen beziehungsweise an der richtigen Stelle einsortiert. Nach dem Zusammenführen ist die Reihenfolge je Quelle streng monoton.

Das Zusammenführen ist eine reine Funktion über Puffer und Nachlieferung.

5. **Speicher und Rendern begrenzen**

   Je Quelle wird ein Ringpuffer fester Größe gehalten (Vorgabe: 600 Werte). Bei sechs Quellen und fünf Werten je Sekunde treffen rund 30 Nachrichten pro Sekunde ein – ein React-Render je Nachricht ist keine Lösung. Sammeln Sie eingehende Werte und schreiben Sie den Zustand gebündelt, höchstens etwa vier- bis zehnmal pro Sekunde. Begründen Sie Ihre Wahl kurz im README.

6. **Darstellung**

   - **Statusübersicht**: Kachel je Quelle mit aktuellem Wert, Einheit, Zustand (ok / Grenzwert / veraltet) und dem Alter des letzten Wertes. Kommt von einer Quelle länger als 10 Sekunden nichts, wird sie als veraltet gekennzeichnet, auch wenn die Verbindung insgesamt steht.
   - **Verlauf**: Liniendiagramm für die gerade ausgewählte Quelle, mit sichtbar markierten Lücken. Bibliothek (z. B. Recharts, visx, uPlot) oder eigenes SVG bzw. Canvas, ganz wie Sie möchten.

3 Anforderungen

- React mit TypeScript, `strict` aktiv, kein `any` und kein `@ts-ignore`. Next.js oder Vite, beides ist recht.
- Transport sind Server-Sent Events (`EventSource`).
- Backoff-Berechnung, Zusammenführen von Strom und Nachlieferung sowie der Ringpuffer liegen in reinen, einzeln testbaren Modulen ohne React-Bezug.
- Nach einer Trennung von 30 Sekunden zeigt der Verlauf anschließend dieselben Werte wie eine durchgehend verbundene Sitzung – keine Lücke, kein doppelter Punkt.
- **Icons ausschließlich aus einem Set**, Lucide oder Phosphor. Kein Mischen, keine Emojis als Symbole.
- **Skeleton-Loader** für Inhaltsflächen (Kachelraster, Diagramm) beim Erstaufbau, **Spinner** nur für kurze Aktionen. Platzhalter in Zielgröße: Weder eintreffende Daten noch ein Wechsel des Verbindungszustands dürfen das Layout springen lassen.
- Tastaturbedienbar, sichtbarer Fokus, Kontrast nach WCAG AA; Zustand nie allein über Farbe. Responsiv ab 360 px, kein waagerechtes Scrollen.
- Tests mit Vitest oder Jest:
  - Backoff: Wachstum, Obergrenze, Streuung in den Grenzen, Rücksetzung erst nach stabiler Verbindung.

- Zusammenführen: Dublette, Werte außer der Reihe, überlappende Nachlieferung, Lücke größer als der Serverpuffer.
- Ringpuffer: Verhalten beim Überlauf.

4 Abgabe

- Ein Git-Repository mit nachvollziehbarer Historie, Frontend und Mock-Server im selben Baum (z. B. `app/` und `server/`). Die Skripte `dev`, `build` und `test` müssen laufen.
- README mit: Startanleitung für beide Teile, Aufbau der Zustandsmaschine, Vorgehen beim Nachziehen der Werte, Wahl der Renderfrequenz, Grenzen.
- Eine Anleitung, wie ein Prüfer einen Abbruch auslöst und was er sehen soll:

```
cd app    && npm install && npm run dev   # Frontend
cd server && npm install && npm run dev   # Mock-Server, Port 4000
curl -X POST localhost:4000/api/steuerung/stoerung \
     -H 'content-type: application/json' -d '{"dauerSekunden":20}'
```

- Keine echten Zugangsdaten, Kundennamen oder Schlüssel im Repository, auch nicht in Testdaten.

5 Bewertungskriterien

- **Korrektheit unter Störung** – Nach mehreren erzwungenen Abbrüchen stimmt der Verlauf: keine Dubletten, keine stillen Lücken, Reihenfolge je Quelle monoton. Das ist das Hauptkriterium.
- **Verbindungsverhalten** – Backoff wächst, ist gedeckelt, streut und setzt an der richtigen Stelle zurück. Der Server wird bei einem längeren Ausfall nicht mit Versuchen überzogen.
- **Ehrlichkeit der Oberfläche** – Was nicht bekannt ist, wird als unbekannt dargestellt. Veraltete Quellen, nicht schließbare Lücken und der Zustand `aufgegeben` sind erkennbar, ohne ins Protokoll zu schauen.
- **Architektur und Leistung** – Netzlogik, Zusammenführen und Darstellung sind getrennt und ohne laufenden Server testbar; das Dashboard bleibt flüssig und wächst im Speicher nicht unbegrenzt. Getestet wird das, was im Betrieb schiefgeht, nicht nur der Pfad, der ohnehin funktioniert.
- **Oberfläche und Dokumentation** – ein Icon-Set, Skeletons statt Spinner für Flächen, kein Layout-Sprung, Barrierefreiheit; das README erklärt die Entscheidungen, nicht nur die Befehle.