Technical Test – Frontend Developer

1. Overview

This test is designed for Frontend Developers applying for a position at [redacted]. Build a small, responsive **Operations Dashboard** in **Next.js** with **two mandatory modules + one optional**. Use a mock API. **No authentication.**
Emphasis on **clean components**, **React Query for server state**, **Formik + Yup** for forms, and **solid loading/empty/error states**.

2. Modules

Mandatory - Assets

- **Table** of assets (sortable, searchable, paginated).
- **Row click → drawer** with basic details (name, site, supplier, risk badge).
- **Vulnerabilities list** inside the drawer (simple list with severity).
- **Filter form** (Formik): search, site, supplier, risk level.

Empty & error states (required):

- Loading skeleton for table and drawer.
- Empty table message with a **"Clear filters"** button.
- Error state with **Retry** and a small **"Show details"** toggle.

Mandatory - Network Topology (React Flow)

- Render a **read-only** topology from mock data:
  - Nodes: **Sites → Gateways → Devices** (3 tiers).
  - Edges: site → gateway, gateway → device.
- Include **MiniMap**, **Controls**, and **FitView**.
- **Selecting a node** shows a small side panel with node name/type and (if device) a link that opens the asset drawer (same detail component as above).
- **Empty & error states:**
  - Loading placeholder.

- Empty graph hint (“No topology available”).
- Error component with Retry.

Optional

Local Notes / Custom Filters (IndexedDB)

For the Notes:

- In the asset drawer, a small **“Notes”** textarea + **Save**.
- Persist notes per asset **in IndexedDB** (client-side only).
- List recent notes in a simple panel.
- Validate with Yup (≤ 200 chars).
- Empty: “No notes yet”. Error: “Couldn’t load notes”.

For the Custom filter:

- In the filter area add a drop-down menu or similar allowing to select between previously saved filters.
- Allow add new filter according with the current selection and a custom name.
- Persist filters **in IndexedDB** (client-side only).

3. Stack & Libraries (keep it lean)

- **TypeScript**
- **Next.js + React**
- **TailwindCSS**
- **React Query (TanStack)** for server state (assets, vulnerabilities, topology, compliance)
- **Formik + Yup** for forms (filters, notes)
- **Axios** for HTTP
- **React Flow** for topology
- **Browser IndexedDB**

4. State & Caching

- **React Query**
  - Query keys reflect filters/pagination.

    - Use placeholderData for table paging.
- **Client/UI state**
    - Small store (Zustand or Context) for: drawer open/close, selected asset ID, selected topology node.
    - **URL ↔ state sync** for table filters and pagination (query string).

5. Empty, Loading & Error

Implement for **every** data-driven surface:

- **Loading**: skeletons that resemble final layout (no jumpy UI).
- **Empty**: clear copy + actionable next step (e.g., Reset filters).
- **Error**: concise message + **Retry** button; small details toggle (status code + endpoint).
- **Form errors**: inline field errors via Yup + a small form banner/toast if submit fails.

6. Documentation (minimal but clear)

Your **README** should include:

- How to run the app and mock API.
- Which optional module you chose and why.
- Where to see **loading/empty/error** states.
- Quick notes on query keys, filter ↔ URL sync, and any IndexedDB usage.
- Known trade-offs & “if I had more time”.

7. Testing (nice-to-have)

A single **Playwright** happy-path is enough:

- Loads assets, applies a risk filter (shows empty when no match).
- Opens a row, displays vulnerabilities.
- (If Notes chosen) saves a note and sees it persist after reload.
- Topology renders with expected node/edge counts.

8. Evaluation

- UX clarity and responsiveness.
- Correct, tidy use of **React Query** (keys, caching) and light UI state.
- Quality of **empty/loading/error** states.
- Basic **React Flow** usage.
- Code organisation, TypeScript hygiene, and README.

9. Mock API (Temporary Database)

To run the mocked api you can use the JSON-SERVER lib
