**FR 1.1 — User Login (Must)**

**What it does:** Lets engineers/managers/admins authenticate with a username and password so we can tie actions to a real person.  
**How it works (DB→BE→FE):** Users live in users with a unique username and a role. The backend exposes /login, checks the credentials, and returns a short-lived token plus the user’s display name. The mobile app calls api.login(...), stores the token **in memory** (not on disk), and navigates to the main menu with the engineer’s name in props.  
**Why it’s a Must:** Almost everything else (roles, auditability, who submitted what) depends on knowing the user. Skipping login would break NFR 3.1 (security) and undercut trust in the data.  
**Why this approach:** Token-in-memory is simple for a prototype and avoids local secret storage. It’s also easy to pass along in the Authorization header for each API call.  
**Impact on the prototype:** Enables session-aware screens, lets us show the logged-in name/role in the header, and lays the groundwork for role-based features later.

**FR 1.2 — User Signup (Must)**

**What it does:** Lets new engineers create accounts without touching the database by hand.  
**How it works:** The Sign-Up screen collects fullName, username, email, and password then calls /register. The server validates, hashes the password, and inserts the row. The app then redirects the user to Login.  
**Why it’s a Must:** If account creation needs an admin/DBA, field use stalls. This unblocks testing and real-world usage.  
**Why this approach:** Server handles validation and hashing; the client stays thin and focuses on UX.  
**Impact:** Faster onboarding, less friction, and consistent user records that match what Login expects.

**FR 1.3 — Show User Name & Role in Header (Must)**

**What it does:** Keeps the UI context-aware—users can see “who’s logged in” at a glance.  
**How it works:** After login, the app passes engineerName (and role if needed) into a shared <AppHeader>. No global state required; props keep it explicit.  
**Why it’s a Must:** It’s a small feature that prevents confusion, helps with accountability, and improves UX.  
**Why this approach:** Simple, readable, and doesn’t over-engineer state management for a prototype.  
**Impact:** Cleaner mental model for users and reviewers—reduces “which account am I using?” mistakes.

**FR 2.1 + FR 2.2 — Select or Insert Site & Zone (Must)**

**What it does:** Engineers can either pick an existing site/zone or type a new one and have it created on the fly.  
**How it works:**

* **DB:** sites uses UNIQUE(site\_name, building\_number); zones uses UNIQUE(site\_id, zone\_name). That stops dupes while allowing the same site name at different buildings.
* **Backend:** “Ensure” endpoints (POST /sites/ensure and POST /zones/ensure) check for an existing row and insert if missing, returning the ID either way.
* **Frontend:** Inputs show typeahead (suggestions). If the user types a new value, the app calls the ensure endpoints before moving on, so foreign keys are valid.  
  **Why it’s a Must:** Real sites/zones pop up in the field. Blocking data entry because a record doesn’t exist yet would kill usability.  
  **Why this approach:** Best of both worlds: flexible free-text UX, but still strict referential integrity through the server.  
  **Impact:** No dead-ends during inspections, cleaner data, and reliable keys for reporting later.

**FR 3.1 + FR 3.2 — Category + Dynamic Item Types (Must)**

**What it does:** The app switches between “Facility” and “Machine Safety,” then lists item types that belong to the chosen category.  
**How it works:**

* **DB:** item\_types stores (inspection\_category, item\_type\_label) with a uniqueness guard.
* **Backend:** /item-types?category=… pulls current options from the DB.
* **Frontend:** When the category changes, the app fetches types and updates the list—no hardcoding.  
  **Why it’s a Must:** Item types change over time. Hardcoding would slow updates and break NFR 5.1 (maintainability).  
  **Why this approach:** It’s data-driven and easy to expand—ops can add a row and the app picks it up.  
  **Impact:** We can scale the catalog without redeploying the app.

**FR 4.1 — Load the Correct Inspection Form (Must)**

**What it does:** Each category gets the right form layout and helper text, while sharing core validation and submission.  
**How it works:** We navigate to category-specific screens (e.g., Facilities) but centralize common rules (e.g., compute overall, comment requirements).  
**Why it’s a Must:** The tasks and wording differ per category, so one giant conditional form would get messy and bug-prone.  
**Why this approach:** Separate components are easier to maintain and reason about; shared logic prevents duplication.  
**Impact:** Lower regression risk and faster iteration when we tweak either form.

**FR 4.2 — Dynamic Subchecks by Item Type (Must)**

**What it does:** The checklist (subchecks) is generated from the database based on the chosen item type, including which checks are **mandatory**.  
**How it works:**

* **DB:** subcheck\_templates stores label, description, value\_type, and sub\_template\_mandatory, linked to item\_types.
* **Backend:** /subcheck-templates?itemTypeId=… (or the by-label variant) returns templates.
* **Frontend:** After the user picks an item type, the app maps templates into UI state (name, valueType, mandatory, etc.).  
  **Why it’s a Must:** This is how we avoid shipping new app versions just to tweak a checklist.  
  **Why this approach:** Templates are the source of truth; the app just renders them.  
  **Impact:** Ops can refine checks centrally and field engineers immediately see updated tasks.

**FR 4.3 — Record Outcomes (Boolean/Numeric) (Must)**

**What it does:** Captures structured results for each subcheck: pass/fail/na and, where needed, a reading.  
**How it works:**

* **DB:** subcheck\_results supports value\_type, result, and optional readings (reading\_number, reading\_text).
* **Backend:** Validates allowed values and normalizes types.
* **Frontend:** MVP uses a simple pass/fail toggle (SubcheckToggleRow). Numeric inputs are already supported by the schema and can be enabled with a small UI enhancement.  
  **Why it’s a Must:** Structured data lets us compute overall results now and do analytics later.  
  **Why this approach:** Start simple with booleans (common case), but keep the door open to numbers without DB changes.  
  **Impact:** Clean data model that’s future-proof for richer measurements.

**FR 4.4 — Optional Comment per Subcheck (Must)**

**What it does:** Lets engineers explain unusual results or leave context.  
**How it works:**

* **DB:** subcheck\_results.comment stores the note.
* **Backend:** Accepts and persists it; we can enforce “comment required on fail” if policy demands.
* **Frontend:** The validation layer supports comments even when the UI hides them by default to keep the form short.  
  **Why it’s a Must:** Comments make reviews much easier and support audits.  
  **Why this approach:** Keeps the main flow fast but captures important context when needed.  
  **Impact:** Better traceability and fewer back-and-forths with managers.

**FR 4.5 — Attach Evidence (Photos/Files) (Must)**

**What it does:** Engineers can attach photos or files—**required** for fails, optional for passes.  
**How it works:**

* **DB:** An attachments table links each file to either the inspection or (preferably) a specific subcheck\_result. It stores a path/URI and an optional caption.
* **Backend:** Validates that failed checks come with at least one attachment, saves metadata, and stays agnostic about storage (local path now; S3/Azure later).
* **Frontend:** Adds an “Attach evidence” action to subchecks. The payload includes file URIs, which the server records.  
  **Why it’s a Must:** It enforces BR2 (“failed items must include a comment and attachment”) and increases trust in the data.  
  **Why this approach:** Decoupling media from the subcheck keeps payloads light and makes it easy to move storage later.  
  **Impact:** Compliance-ready records and fewer disputes about “what was seen” on site.

**FR 4.6 — Automatic Overall Result (Must)**

**What it does:** The system computes PASS/FAIL automatically based on mandatory subchecks.  
**How it works:**

* **DB:** inspections.overall\_result stores the outcome.
* **Backend:** Builds a mandatory map from templates and evaluates the submitted subchecks: any failed **mandatory** subcheck → overall fail. The client can preview, but the server is the source of truth.  
  **Why it’s a Must:** Prevents mistakes and enforces BR5 consistently.  
  **Why this approach:** Centralizing the rule means all clients behave the same and it’s easy to evolve (e.g., category-specific logic).  
  **Impact:** Consistent outcomes and simpler QA—no manual toggles to forget.

**FR 4.7 — Submit & Save Inspection (Must)**

**What it does:** Persists a complete inspection reliably (no half-saved forms).  
**How it works:** The backend runs a **single transaction**: ensure site/zone/item exist (using the ensure endpoints), compute overall, insert the inspection row, insert subcheck results (and attachments), then commit. Any validation error (e.g., missing required evidence) rolls back the whole thing.  
**Why it’s a Must:** Data integrity is everything; broken foreign keys or partial saves make reports useless.  
**Why this approach:** Transactions guarantee atomicity and keep the database consistent even when the network or input is flaky.  
**Impact:** Trustworthy records, easier debugging, and a solid foundation for reporting (the deferred FR 5.x).

**Big picture: why this set of choices works**

* **Data-driven UI:** Item types and templates live in the DB (NFR 5.1), so the app adapts without re-releases.
* **Thin client, smart server:** Critical rules (overall result, evidence on fail) are enforced server-side so all clients behave the same.
* **Prototype-friendly, production-leaning:** Token auth in memory, “ensure” endpoints, and transactional writes are simple enough to ship now and strong enough to grow later.
* **User-first workflows:** Free-text + typeahead + ensure avoids blocking engineers in the field while still keeping the data clean.

If you want, I can convert this into a short appendix table (Requirement → Implementation notes → Trade-offs → Impact) to drop straight into the report.