PMO RAID Register

The spec, the plan, and the constitution

A spec-driven, agentic delivery project. These three documents define what the tool does, how it gets built, and the non-negotiable rules the AI must follow.

Author: Apurv Duhan, PMP

Created 14 June 2026

THE SPEC

What the PMO RAID Register Is, and What It Needs to Do

What this document is: This is the plain description of the tool — what it's for and what it has to do — written so anyone can read and approve it without needing a technical background. A separate document covers how it gets built. This one stays focused on the what and the why.

Status: Ready to build (all open questions answered, see section 5)

Author: Apurv Duhan, PMP

Last updated: 14 June 2026

---

1. The problem I'm solving

Every project has four things a manager constantly has to watch: Risks (what could go wrong), Assumptions (what we're taking for granted), Issues (what's already gone wrong), and Dependencies (what we're waiting on from someone else). Together these are called a RAID log.

Most teams keep this in a spreadsheet that gets messy fast. There's no record of who changed what, ownership gets fuzzy, and pulling together a clean update for leadership means rebuilding the whole thing by hand every week.

This tool fixes that. It gives a project manager one place to log these items, keep them current, see the full history behind every one of them, and print a tidy status summary on demand.

---

2. Who uses it and how

One project manager, on their own computer. There's no sign-in and no shared server in this first version, because only one person uses it. Every change is recorded under a display name the PM sets once when they start using the tool.

---

3. What it should do, walked through

Here are the main things a PM should be able to do, described as they'd actually happen:

Logging a new item. The PM adds an item, picks its type (Risk, Assumption, Issue, or Dependency), gives it a title and description, names an owner, and sets how serious it is.

The tool gives it a readable ID like **`RISK-001`**, marks it "Open," and records that it was created.

Updating an item without losing the past. The PM changes an item's status or edits its details. The item shows the new information, but its history still shows what it said before, who changed it, and when.

Finding things quickly. The PM filters the list by type and status — say, only open dependencies — and sees a count of what matches.

Producing a status report. The PM exports a single printable summary, grouped by type, showing each item's current status, owner, and severity, stamped with the date and time it was produced.

Removing something safely. The PM deletes an item, but only after confirming they mean to, and the deletion is recorded in the history.

A few edge cases worth stating plainly: an item with no title is refused with a clear message; exporting an empty list still produces a valid report that simply says nothing's been logged yet; and a bad entry (like an invalid severity) is refused without wiping out the rest of what the PM typed.

---

4. The specific requirements

These are the testable requirements. Each is numbered so that, later, every screen and every test can point back to the exact one it satisfies. That numbered thread is what lets me prove the finished tool actually does what was agreed.

1. Let the user create an item with a type, title, description, owner (name and email), severity, and status, choosing type, severity, and status from set lists (see requirement 13).
2. Give every item a unique, readable ID when it's created.
3. Support a sensible set of statuses for an item's life cycle (from requirement 13).
4. Let the user edit any changeable detail of an existing item.
5. Record every create, edit, and delete in a permanent history that can't be altered — capturing the time, who did it, and the before-and-after values. The "who" is the local display name (requirement 12).
6. Show an item's full history whenever the user asks for it.
7. Let the user filter the list by type and status, with a live count of what matches.
8. Only delete an item after the user explicitly confirms.
9. Export one printable status report grouped by type, showing current status, owner, and severity, stamped with the date and time.

10. Refuse invalid entries with a clear, plain message, without discarding the user's other valid input.
11. Never rely on color alone to show an item's status, so it's readable for colorblind users.
12. Use one display name the user sets locally as the "who" for all history; no passwords or accounts in this version.
13. Use these set lists, fixed for now:

- Type: Risk, Assumption, Issue, Dependency
- Severity: Low, Medium, High, Critical
- Status: Open, In Progress, Mitigating, Resolved, Closed

---

5. Questions I answered before building

Early on, three things were genuinely open. Rather than let the tool guess, I settled them on the record:

- One user or many? One, local, no sign-in for this version. Multiple users can come later.
- One export format or several? One printable report. A data-file export can come later.
- Should the type/severity/status lists be editable? No, they're fixed for now. Making them editable can come later.

---

6. What this version deliberately leaves out

So expectations are clear, this first version does not include: tracking multiple projects at once, user accounts or permissions, connections to other project-management tools, email or notifications, or two people editing at the same time. Each of those is a deliberate "later," not an oversight.

---

7. Sign-off checklist

- All the open questions are answered (section 5)
- Every requirement can be tested
- Every requirement traces to something in section 3 or a project rule
- No technical "how" has crept into this document — it stays about the "what"
- The locked history covers every change, with no exceptions
- A non-technical stakeholder has read this and approved it

THE PLAN

How I'm Building the PMO RAID Register

Author: Apurv Duhan, PMP | Last updated: 14 June 2026

What this document is: The spec (the other file) covers what this tool does and why. This one covers how I plan to build it — the tools I picked, how the information is organized, what screens and buttons exist, and the order I'll build things in.

It has two readers in mind. The first is the AI coding agent that will actually write the code: the clearer my plan, the closer the result matches what I intended, instead of the agent guessing. The second is any person reviewing my work — a teammate or a hiring manager — who can read this and see that I made deliberate decisions up front rather than letting the AI improvise.

---

1. The short version of what I'm building

A small web app that one project manager runs on their own computer. It tracks the four things every project has to keep an eye on — Risks, Assumptions, Issues, and Dependencies — keeps a full history of every change, and prints a clean status summary to share with leadership.

---

2. The choices I made, and why

I kept every choice as plain and boring as I could, on purpose. One of the project's core rules is "use the simplest thing that does the job," so I deliberately avoided anything fancy.

- It runs on the PM's own machine, with no login. Only one person uses this tool, so building accounts, passwords, and a hosted server would be effort spent solving a problem I don't have. If a future version needs multiple users, that's when I'd add it.
- All the data lives in a single file on that machine. I'm using SQLite for this, which is a tiny, built-in database that needs zero setup. For one person tracking a few hundred items, it's more than enough and there's nothing to install or maintain.
- The pages are plain web pages, not a heavy app. I'm building it in Python with a lightweight web framework (FastAPI) and simple HTML pages. Fewer moving parts means it's easier to keep accessible for everyone, and nothing on the page tracks the user or calls out to the internet.
- The change history is locked. Every edit gets written to a separate history record that the app is never allowed to alter or erase. I'm backing that up at the database level too, so even a future coding mistake can't quietly rewrite the audit trail. This is the part that makes the whole "auditable" promise real, so it's not optional.

---

3. Checking the plan against the project's own rules

Before any code gets written, I check this plan against the non-negotiable rules I set for the project (those live in the constitution file). Here's how the plan honors each, in plain terms:

- Everything traces back to a requirement. Every screen, button, and test points back to a numbered requirement from the spec, so I can always show the line from "what we agreed to build" to "here's the code and the test that proves it."
- Nothing changes without a record. Every create, edit, and delete writes a history entry, and that history can't be altered.
- Tests come first. For each thing the tool should do, I write the check that proves it works before writing the code that does it.
- Privacy stays tight. The tool only stores an item owner's name and email. No analytics, no trackers, no data leaving the machine.
- It's usable by everyone. The pages work with a keyboard, have clear labels, and never rely on color alone to show status (so it's readable for colorblind users).
- Problems are never silent. If something goes wrong, the user sees a plain-English message and the technical detail gets logged.

If any choice had broken one of these rules, I'd fix the plan before building — not patch it afterward.

---

4. How the pieces fit together

The information. The tool keeps two kinds of records:

1. RAID items — the actual things being tracked. Each one has a type (Risk, Assumption, Issue, or Dependency), a title and description, an owner (name and email), a severity (Low to Critical), a status (Open, In Progress, Mitigating, Resolved, Closed), and the dates it was created and last changed. Each gets a readable ID like **`RISK-001`**.
2. History entries — a permanent, unchangeable log. Each entry records what changed, who changed it, when, and the before-and-after values.

The screens and actions. Below is everything the tool can do. The numbers in brackets are the requirement each action satisfies — that's the traceability thread I mentioned, and it's worth keeping because it's exactly what proves the work was built to spec.

- See the full register, with filters for type and status and a live count (req. 7)
- Add a new item, which assigns its ID and logs the creation (reqs. 1, 2, 5)
- Open a single item and read its full change history (req. 6)
- Edit an item, logging each field that changed (reqs. 4, 5)
- Delete an item, but only after a confirmation step, and the deletion is logged (reqs. 8, 5)

- Export one printable status report, grouped by type and stamped with the date and time (req. 9)

Anywhere the user can type something in, the tool checks the input and, if it's wrong, shows a clear message without throwing away the rest of what they entered (reqs. 10, 11, 12).

---

5. The order I'll build it in

I won't write the task-by-task list here — that's the next step. But the order follows a few simple rules: the data and the locked history come first, then the input checks, then the screens, then the export, and finally a pass to confirm it's accessible and that errors are handled cleanly. For each feature, the test that proves it works gets written before the feature itself.

---

6. What I'm deliberately keeping simple

I didn't have to bend any of the project's rules to make this plan work, and I didn't add anything the spec didn't ask for. If a future need forces something more complicated (multiple users, for example), I'll write down why here and update the project rules first — so even the decision to add complexity is on the record.

---

7. Where things stand

- Spec finished, no open questions
- Plan checked against the project rules — all clear
- Decisions and structure written down (this document)
- Break the plan into a task list
- Final consistency check across all the documents
- Build it

THE CONSTITUTION

Project Constitution — PMO RAID Register

> The non-negotiable principles that govern every plan, task, and line of code in this project. The AI coding agent MUST adhere to these articles. Any plan that violates an article must be revised before implementation. When an article and a feature request conflict, the article wins until formally amended below.

Version: 1.0.0

Ratified: 2026-06-14

Author: Apurv Duhan, PMP

Article I — Spec-First & Traceability (NON-NEGOTIABLE)

No code is written without a corresponding requirement in **`spec.md`**. Every functional requirement carries a stable ID (FR-XXX). Every task and every test references the requirement ID it satisfies. The result is a continuous trace from intent spec task test, reconstructable at any time. If a request has no requirement, the spec is amended first.

Article II — Auditability of Data Changes (NON-NEGOTIABLE)

Every create, update, or delete to a RAID item MUST be recorded in an append-only audit log capturing: timestamp (UTC), actor, item ID, field changed, old value, new value. Audit records are never editable or deletable through the application. Absence of an audit trail for any data mutation is a defect.

Article III — Test-First

Acceptance scenarios in **`spec.md`** (Given/When/Then) are translated into automated tests before the implementing code exists. A feature is "done" only when its referenced tests pass. No requirement ships without at least one test mapped to its FR ID.

Article IV — Simplicity & No Premature Architecture

Start with the simplest design that satisfies the spec. No microservices, no message queues, no caching layer, no third-party dependency unless a requirement demonstrably needs it. Every added dependency must be justified in the plan against this article.

Article V — Data Privacy & Minimization

Collect only the data the spec requires. No personally identifiable information beyond a display name and email for the item owner. No analytics, telemetry, or third-party trackers.

Data stays within the application's own datastore.

Article VI — Security Baseline

No secrets, keys, tokens, or credentials in source code or commit history — environment variables only. All user input is validated and treated as untrusted. The application performs no destructive action without explicit user confirmation in the UI.

Article VII — Accessibility

The interface meets WCAG 2.1 AA: full keyboard operability, visible focus states, sufficient color contrast, and labels on all interactive elements. RAID status MUST NOT be conveyed by color alone (use text/icon + color).

Article VIII — Observability

The application logs meaningful events (startup, errors, data mutations) in a structured, machine-readable format. Errors surface a clear, non-technical message to the user and a detailed entry to the log. Silent failures are prohibited.

Article IX — Plain-Language Stakeholder Readability

`spec.md` MUST remain readable by a non-technical PMO stakeholder. Implementation detail belongs in the plan, never the spec. If a stakeholder cannot understand what the product does from the spec alone, the spec has failed.

Governance

- This constitution supersedes all other practices. Plans are checked against every article during the **`/analyze`** quality gate.
- Amendments require: a documented rationale, a version bump (semantic), and an updated ratification date. Amendments are committed as their own change so the governance history is itself auditable.
- Reviewers (human-in-the-loop) verify article compliance at plan review — before implementation, not after.

Compliance is not a final checklist; it is the condition under which the agent is permitted to work.
