Quarterly Project Report #1

RetailSync

What Did You Do

This quarter, I worked on building the foundation of **RetailSync**, a multi-tenant retail operations platform designed for small grocery stores and gas stations. The goal of the project is to solve a very practical problem: many small retail businesses rely on spreadsheets, exported POS reports, and manual reconciliation to manage inventory and finances. There is often no structured system connecting sales data, inventory movement, supplier invoices, and bank deposits.

RetailSync aims to centralize these workflows into a single system.

So far, I have implemented:

- A multi-tenant architecture where each company's data is isolated using a `companyId`
- Authentication using JWT access and refresh tokens
- Role-based access control (Admin, Member, Viewer)
- POS CSV import functionality with daily and monthly reporting
- An event-sourced inventory system using an immutable ledger model
- Docker support for consistent local development

The intended audience includes small retail store operators who need structured operational tools without enterprise-level complexity. From a technical perspective, the project also demonstrates SaaS architecture patterns such as tenant isolation, RBAC enforcement, and ledger-based data modeling.

The tech stack includes:

- **Frontend:** React, TypeScript, Redux Toolkit, Material UI (Vite-based)
- **Backend:** Express, TypeScript, MongoDB, Mongoose, Zod, JWT
- **Testing:** Vitest
- **DevOps:** Docker Compose
- **Monorepo:** pnpm workspace structure

I chose this stack because it provides full-stack type safety, modularity, and scalability.

How Did You Do It

Before starting implementation, I researched common SaaS design patterns, especially multi-tenant architectures and role-based permission systems. I also studied inventory modeling approaches and chose an event-sourced pattern instead of storing a mutable "current quantity" field.

In RetailSync, inventory changes are recorded as append-only ledger entries (`InventoryLedger`). Current stock levels are calculated using aggregation. This approach improves traceability and prepares the system for future audit or reconciliation features.

I also researched:

- JWT authentication with refresh token patterns
- Server-side RBAC enforcement strategies
- MongoDB aggregation pipelines
- Monorepo management with pnpm
- Docker-based development workflows

Pros of the Tech Stack

- TypeScript across client and server improves reliability.
- MongoDB aggregation works well for ledger-based calculations.
- pnpm workspaces enforce clean dependency boundaries.
- Docker makes the development environment reproducible.

Cons / Tradeoffs

- Event-sourced inventory adds query complexity.
- MongoDB transactions require careful handling for financial flows.
- Strict multi-tenant enforcement requires disciplined query design.
- Future reconciliation logic will increase algorithmic complexity.

I also discussed architecture decisions with classmates, especially around permission modeling and Docker configuration. Those conversations helped refine RBAC enforcement and clarify how client-side permission gating complements server-side enforcement.

What Problems Did You Encounter

One of the main challenges was ensuring strict tenant isolation. Every query — including aggregation pipelines — must filter by `companyId`. Missing that in even one place could cause cross-tenant data exposure, which is unacceptable in a SaaS system.

Another challenge was designing the event-sourced inventory model. It required rethinking how stock should be represented. Instead of updating a quantity directly, everything is derived from ledger entries. While this improves traceability, it increases complexity in aggregation and testing.

On the infrastructure side, I encountered DNS resolution errors during `pnpm install` while running a full CI-equivalent local check. This turned out to be an environment/network issue rather than a code issue, but it temporarily blocked dependency installation. I also encountered a Docker BuildKit storage metadata error during one build attempt, which required local environment cleanup.

Concerns Moving Forward

- Financial reconciliation (Phase 4) will require carefully designed matching algorithms.
- Invoice confirm flows must be transaction-safe to avoid duplicate writes.
- File uploads currently use local storage; production storage needs to be formalized.
- As features expand, indexing and performance tuning will become more important.

Hardware / Cloud Needs

Currently, the project runs locally using MongoDB and Docker without special hardware requirements.

Future phases may require:

- Object storage (e.g., S3-compatible service) for file uploads
- A hosted MongoDB instance for staging/production
- CI resource tuning once E2E testing is introduced

Summary

This quarter focused on building a strong architectural foundation: authentication, tenant isolation, RBAC, POS ingestion, and event-sourced inventory.

The system is now structurally stable and ready for more advanced workflows such as invoice ingestion and financial reconciliation. The next phase will increase complexity, but the foundational design decisions made this quarter position the project for scalable growth.