SmartStock System Design Document

EECS 3311 — Sprint 2

Group: Project JUME (Group 9)

Date: April 3, 2025

Table of Contents

1. Introduction

2. CRC Cards

• 2.1 Frontend CRC

• 2.2 Backend CRC

3. Software Architecture

4. Error Handling Strategy

5. Conclusion

1. Introduction

InventoryFlow is a modern and user-friendly inventory management system built to help teams organize products, track orders, and manage customers across multiple organizations. It leverages **Supabase** as a backend-as-a-service for database storage, authentication, and file handling, while maintaining a responsive and elegant frontend using **Next.js**, **ShadCN UI**, and **TanStack Table (table.sadmn)** for powerful data tables.

The system is built with robust error handling and validation in mind. All form submissions, such as creating a product or order, are validated both on the **frontend** using **React Hook Form + ZUD**, and on the **backend** via Supabase constraints and Row-Level Security (RLS) policies. This ensures data integrity and a smooth user experience.

We used **Supabase Auth with OAuth (Google and GitHub)** for secure login, and implemented a full organizational model with role-based access control (admin, manager, staff). Our PostgreSQL schema includes well-defined relationships, enums, triggers, and utility functions to support scalable, real-time collaboration.

The goal was to build something clean, extensible, and production-ready. A key milestone in this version was implementing **tables, forms, editing/searching capabilities with role-based access control for admins, managers, and staff, with addition to Multi-tenancy of each user to freely switch between entirely different warehouses.**

---

2. CRC Cards

The system architecture is divided between a structured backend and a clean, user-focused frontend. The backend is responsible for securely storing and managing all data related to users, organizations, inventory, and orders. It enforces strict access control and handles business logic like role permissions, triggers, and stock notifications. The frontend, on the other hand, is responsible for capturing user input, displaying inventory and order data, and handling real-time interactions.

At the core of the backend is the **database**, which organizes all records across various entities including users, organizations, members, products, orders, and restock notifications. It ensures security and consistency through policies, triggers, and constraints. Every action that affects data flows through this system, whether it's creating a new customer or fulfilling an order.

The **users** section of the system manages personal profiles. When a user signs in, their basic information is stored and synced, allowing the app to recognize them and enforce access rules. Each user can only access their own data unless they are granted additional roles through an organization.

The **orgs** component represents a business or team. It stores contact and address details and serves as the primary boundary for isolating data between different organizations. It's the anchor point for linking all other data in a multi-tenant setup.

org_members connects users to organizations and assigns them a role. This relationship is critical for enforcing what a user can see or do. For example, only users with admin or manager roles can modify data, while others may only view it.

The **customers** component holds data about external clients tied to an organization. These records include things like contact information and delivery addresses. Customers are linked to orders, creating a trail from who placed the order to what was purchased.

The **products** section is where inventory details are managed. It includes information like product names, quantities, prices, and stock thresholds. When inventory levels fall below a defined point, it can trigger automated alerts to notify the organization.

orders capture purchase activity. Each order is associated with a customer and a product and includes fulfillment status, totals, and timestamps. Orders are restricted so that only authorized users in the correct organization can create or update them.

Lastly, **product_restock_notifications** are generated automatically when stock levels fall below their thresholds. These alerts are organization-specific and ensure that low inventory is flagged and visible only to the appropriate team members.

On the frontend, each part of the user interface is connected to the backend through structured requests. When users create, update, or view data, they receive clear feedback, including error messages when access is denied, data is missing, or something fails. The interface is designed to be responsive and helpful, making inventory and order management as smooth as possible.

2.1 Frontend Components

SupabaseClient	
Parent Component: none	
Responsibilities: • Interface with Supabase for all data fetching and mutations • Handle authentication and table queries	**Collaborators:** • HomePage, OrdersPage, RecentActivity, ProductsPage, UserService

OrdersPage	
Parent Component: none	
Responsibilities: • Display list of customer orders • Provide modal to create new orders • Allow updating order status and notification fields	**Collaborators:** • SupabaseClient, Table, ModalForm, Sidebar

HomePage	
Parent Component: none	
Responsibilities: • Fetch and display stock summary data • Render pie chart with visual analytics • Display weather and recent activity widgets	**Collaborators:** • Sidebar, PieChartComponent, RecentActivity, WeatherWidget, SupabaseClient

Sidebar	
Parent Component: none	
Responsibilities: • Provide persistent navigation between app sections • Display user profile avatar and link to profile page	**Collaborators:** • Link, UserService, ProfilePag

ProductsPage	
Parent Component: none	
Responsibilities: • Display product listings • Support filtering and searching of products • (Optional) Allow product creation and editing	**Collaborators:** • SupabaseClient, Sidebar, SearchBarComponent

RecentActivity	
Parent Component: none	
Responsibilities: • Fetch and display recent updates to orders (status, notification) • Present activity logs in reverse chronological order	**Collaborators:** • SupabaseClient, OrdersTable

WeatherWidget	
Parent Component: none	
Responsibilities: • Fetch and display current weather conditions from API • Handle loading and fallback states gracefully	**Collaborators:** • Weather API, HomePage

PieChartComponent	
Parent Component: none	
Responsibilities: • Render visual summary of order stats using Recharts • Display chart with interactive hover effect	**Collaborators:** • HomePage, Recharts Library

OrderService	
Parent Component: none	
Responsibilities: • Insert and update orders in the orders table • Retrieve order data based on filters or sort parameters	**Collaborators:** • SupabaseClient, OrdersPage, RecentActivity

2.2 Backend/Service Layer

customers	
Parent Class: SQL Table **Subclasses:** none	
Responsibilities: • Represent customers of an org (contact + address info) • Serve as a link in the org–customer–order–product chain • Support role-based RLS access per organization	**Collaborators:** • orgs: Parent org • orders: Reference customers in transactions • policy.customers_*: Insert/update/delete rules

products	
Parent Class: SQL Table **Subclasses:** none	
Responsibilities: • Store product metadata (name, price, stock, etc.) • Monitor inventory levels and trigger notifications • Track ownership (org/customer), category, SKU	**Collaborators:** • customers: Product recipient • product_restock_notifications: Trigger on low inventory • policy.products_*: Access and role-based controls

UserService	
Parent Component: none	
Responsibilities: • Retrieve user information including name and avatar • Handle profile logic and role validation	**Collaborators:** • Supabase Auth, Sidebar, ProfilePage

ProductService	
Parent Component: none	
Responsibilities: • Fetch product records from the products table • (Optional) Create or update product entries	**Collaborators:** • SupabaseClient, ProductsPage

Database (Main connection)	
Parent Class: Supabase/PostgreSQL **Subclasses:** users, orgs, org_members, customers, products, orders, product_restock_notifications	
Responsibilities: • Store and manage structured data (notifications, users, products, orders, customers, etc.) • Provide centralized access for all CRUD operations • Enforce security via RLS (Row-Level Security) policies • Enable support functions (e.g. moddatetime, UUID generation, inventory triggers) • Enable scheduled jobs and helper functions via extensions	**Collaborators:** • Supabase Client SDK: Executes CRUD and auth logic • Supabase Auth: Triggers user sync into `users` table • Triggers & Policies: Handle data integrity and access control

Users	
Parent Class: SQL Table **Subclasses:** none	
Responsibilities: • Store user profile data synced from Supabase Auth • Provide personal info like name, email, avatar • Enforce RLS policies for per-user access control	**Collaborators:** • auth.users: Supabase auth table that triggers insert • public.handle_new_user(): Function syncing auth metadata • policy.users_*: RLS functions/policies

orgs	
Parent Class: SQL Table **Subclasses:** none	
Responsibilities: • Represent organizations with address, contact, and financial data • Define org-wide data separation (for multitenancy) • Provide org identifiers for other tables (e.g., orders, products)	**Collaborators:** • org_members: Links users to orgs and defines roles • policy.orgs_*: RLS functions for view, update, delete

orgs_members	
Parent Class: SQL Table **Subclasses:** none	
Responsibilities: • Link users to orgs with roles (`admin`, `manager`, `staff`) • Enforce permissions based on roles (RLS policies) • Track membership history (with timestamps)	**Collaborators:** • users: Member identity • orgs: Organization reference • policy.org_members_*: Access control rules

Product_restock_notifications	
Parent Class: SQL Table **Subclasses:** none	
Responsibilities: • Store alerts when product stock drops below a threshold • Avoid duplicate notifications for the same product • Allow only authorized org members to view/delete alerts	**Collaborators:** • products: Triggers insert via handle_product_restock_notification • policy.product_restock_notifications_*: Access control logic

orders	
Parent Class: SQL Table **Subclasses:** none	
Responsibilities: • Record customer purchases with associated products • Store totals, fulfillment status, and notification status • Support full CRUD and RLS per org & role	**Collaborators:** • products: Product reference • customers: Customer placing the order • policy.orders_*: Access control logic

3. Software Architecture

Inventory Software Architecture – Overview

The architecture is structured in three main layers: **Frontend**, **Backend (API)**, and **Backend (DB/Storage)**. These layers interact through clearly defined flows to deliver secure, modular, and efficient functionality.

Frontend (User Interface)

This is the **UI that users interact with**, where they:

- Log in using **Google or GitHub**
- View or manage their **products**, **orders**, **customers**, or **notifications**

Your frontend runs on **port 3000**, and it sends queries to Supabase via the **PostgREST API** or the **Auth/Storage APIs**.

Backend (API Layer)

Supabase Client (PostgREST)

- Acts as the main connector between the frontend and your database.
- Automatically exposes **CRUD endpoints** for your database tables like:
    - `products`, `orders`, `customers`, `org_members`, etc.
- You query data using functions like `.select()`, `.insert()`, etc.
- Under the hood, this uses **PostgREST**, which converts your table schema into RESTful APIs.

Auth Service

- Handles **OAuth login using Google and GitHub**.
- When a user signs in, Supabase creates a user in `auth.users` and automatically syncs it to your custom `users` table using the `handle_new_user()` trigger.

- RLS (Row-Level Security) ensures that:
    - Users can only view or update their own data
    - Org admins/managers have broader access based on their role

Storage API

- Used for handling file uploads (e.g., product images).
- It connects to Supabase Storage Buckets, which function like Amazon S3.
- Files are stored securely and linked to products via image URLs.

Backend (Database & Storage Layer)

PostgreSQL Database

Your app uses **Supabase PostgreSQL**, with:

- **Tables:** `users`, `orgs`, `products`, `orders`, `customers`, `org_members`, `product_restock_notifications`
- **Custom ENUM types:** (`org_role`, `product_status`, `fulfillment_status`)
- **Triggers:** like `moddatetime` for `updated_at` timestamps
- **Utility Functions:** like `prevent_modifying_columns()` to lock critical fields
- **Full RLS Policies:** Every table enforces **fine-grained access control**, based on:
    - `auth.uid()` identity
    - Role (admin, manager, staff)
    - Organization membership

Storage Bucket / S3

Used for saving product-related files (images, docs, etc.). Access is controlled by Supabase's storage schema and custom policies that ensure:

- Only authorized users can upload/view/delete files

- File paths map cleanly to org/product IDs

Supabase Admin Studio

Used during development for:

- Creating tables, types, and functions
- Debugging RLS and triggers
- Browsing data without needing a separate Adminer tool

Data & Flow Summary

1. **User signs in via Google/GitHub**
   Auth service triggers user creation in the database

2. **User performs actions in the UI** (e.g., add product, update order)
   Supabase Client sends REST calls via PostgREST

3. **Database handles logic**
   Triggers + RLS + Enums keep everything secure and consistent

4. **If files are uploaded**
   Storage API handles it and links it to database entries

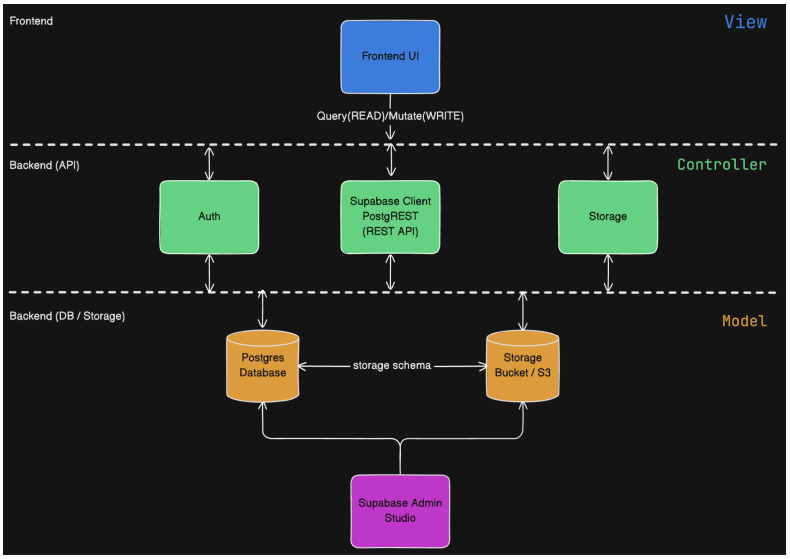

Figure 3.1 Software Architecture Diagram

---

4. Error Handling Strategy

Error handling was a central priority in the development of SmartStock to ensure the system is **resilient**, **secure**, and **user-friendly**. A layered approach was taken across the **frontend**, **backend**, and **database**, combining proactive validation, graceful fallback mechanisms, and meaningful feedback for end users. By leveraging Supabase's features alongside thoughtful UI design, the app avoids silent failures and keeps users informed with clarity and consistency.

Frontend (Client-Side)

The frontend prioritizes **clear communication** and a **smooth user experience** in the face of errors. Users are immediately notified when something goes wrong without disrupting their workflow.

- **Toast Notifications & Alerts:** All asynchronous operations such as form submissions, fetches, or mutations are wrapped in a toast system that surfaces both success and failure states with helpful, visually styled messages.

- **Form Validation:** Using schema validation with React Hook Form and Zod, all inputs (e.g., Create Order, Create Product) are validated **before** API calls are made, reducing invalid requests and improving UX.

- **Error Boundaries:** Rendering errors in specific components are caught using React-based error boundaries to prevent full-page crashes.

- **Fallback UI Rendering:** If a component like a table, chart, or activity feed fails to load or returns no data, the system gracefully displays placeholders or empty states instead of breaking the UI.

- **Permission Handling:** Errors like unauthorized access (due to RLS restrictions) are caught and translated into clear messages, so users are informed when they don't have access to a resource.

- **Retry Logic (Optional):** For transient issues such as failed fetches from third-party sources (e.g., weather), lightweight retry logic is built in to improve reliability.

Backend (Supabase & API Layer)

The backend is powered by Supabase and relies on **PostgREST** APIs and **RLS policies** for security. Backend-side error handling ensures invalid operations are blocked and meaningful errors are returned.

- **Custom RLS Policies:** Every table (e.g., products, orders, org_members) is protected by row-level policies that return clear errors when a user lacks the right permissions (e.g., "Only admins can update this record").

- **Supabase Response Guarding:** All operations are checked for `.error` responses. These errors are logged and surfaced to the frontend for better user awareness.

- **Custom Exception Functions:** Utility functions like `prevent_modifying_columns()` raise specific exceptions for restricted field edits, ensuring consistent logic enforcement.

- **Role-based Access Control Errors:** Queries are scoped to the authenticated user's org and role. Unauthorized access attempts trigger permission-denied errors with user-friendly messages.

- **Rate Limiting (Planned):** A future enhancement includes adding rate-limiting rules on sensitive endpoints to prevent abuse or spam.

- **API Monitoring (Planned):** Integration with uptime or alerting tools (like UptimeRobot) is planned to track backend service health and downtime.

Database-Level (PostgreSQL via Supabase)

The database is designed to **enforce integrity at all times** and prevent invalid or duplicate data from ever being stored.

- **Schema Enforcement:** Tables are structured with `NOT NULL` constraints, enums (e.g., `product_status`, `fulfillment_status`), and foreign key relations to guarantee data consistency.

- **Triggers:** Functions like `moddatetime` automatically update timestamps on every row change, preventing stale data.

- **Unique Constraints:** Keys like `email` and `imdb_id` are protected from duplication, especially important in multi-tenant org setups.

- **Validation with Exceptions:** Custom Postgres functions validate input and raise errors that bubble up to the Supabase client, allowing meaningful handling at the UI level.

- **Staging + Seed Data:** A dedicated staging environment was created with seeded mock data to safely test workflows, simulate edge cases (e.g., low inventory), and verify error messages.

Debugging & Reliability

To support development and ongoing debugging:

- **Supabase Admin Studio** is actively used to inspect RLS policies, run test queries, monitor API responses, and verify triggers.

- Developers can easily validate logic by simulating user roles and permissions with seeded data to replicate real-world use cases and error conditions.

---

5. Conclusion

The second sprint of SmartStock represents a strong foundation for building a secure, scalable, and user-focused inventory management system. With Supabase at the core of the backend, the platform now offers structured data access, automated stock alerts, and a robust role-based permission system through tightly enforced RLS policies. Core database components like users, organizations, orders, products, and notifications have been clearly defined and are working together seamlessly to support real-time, multi-tenant operations.

On the frontend, careful integration with backend logic ensures that all interactions, from inventory updates to customer orders, are intuitive and error-resistant. With thoughtful validation, dynamic data handling, and graceful error feedback, the system maintains a smooth experience even under complex access conditions or unexpected failures.

By structuring the project around clear responsibilities and leveraging modular components across both the UI and backend, SmartStock is positioned to grow into a powerful, production-ready platform. With this solid architecture in place, the next steps will focus on expanding capabilities with features like advanced reporting, permission-based dashboards, and smarter inventory forecasting tools.

Prepared by: Team JUME — Max, Erfan, Jay, Usman