# NovaPOS User Management System: Current State & Implementation Plan

## Current User Management Architecture and Capabilities

*Figure: High-level NovaPOS user management architecture. An* *Auth Service* *(Rust-based) handles employee accounts (users) with its own database, while a separate* *Customer Service* *manages customer records. The Admin Portal (React) communicates with these microservices via REST APIs (with tenant scoping), and future audit logging could flow to a central log store or Kafka.*

**Backend (Auth Service – Employee Accounts):** NovaPOS employs a dedicated **auth-service** microservice for managing employee user accounts (admins, managers, cashiers, etc.). This service exposes endpoints for user authentication and basic user CRUD. Currently, the auth-service supports: creating new users (via POST /users), listing users (GET /users), retrieving role catalog (GET /roles), as well as login/logout and session refresh routes[[1]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/main.rs#L134-L141). User data is stored in a Postgres table keyed by a UUID, with fields for tenant ID (to enforce multi-tenancy), name, email, role, and a hashed password[[2]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L363-L371)[[3]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L393-L401). The system uses Argon2 for password hashing and supports account lockout (fields for failed login attempts and locked\_until timestamp) and optional MFA, as seen in the user schema (e.g. failed\_attempts, locked\_until, mfa\_secret columns added via migrations). **However, critical capabilities are missing**: there is **no API to update an existing user’s information or role, no dedicated endpoint to deactivate/activate accounts, and no password reset functionality** in the current implementation. The auth-service does not define any PUT/PATCH /users/{id} or similar route – only creation and listing are present[[1]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/main.rs#L134-L141). Likewise, there is no is\_active field in the user schema yet to mark accounts as disabled. If an employee leaves or changes roles, admins currently cannot reflect that via the API (beyond possibly deleting and re-creating, which is not ideal). Password recovery flows are also not implemented – there’s no “forgot password” or admin-triggered reset endpoint at this time. In short, **the backend’s user lifecycle coverage is limited to create and read**, with no update or delete (deactivation) endpoints, and only implicit lockout for security but not intentional deactivation. This aligns with internal analysis noting that no explicit update API exists yet and suggesting a need to add one (e.g. a PUT /users/:id to change roles or disable a user)[[4]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L447-L455).

For customer accounts, NovaPOS has a separate **customer-service** microservice to manage **customer profiles** (loyalty accounts, contact info). It stores customer data (name, email, phone, etc.) per tenant, with sensitive fields encrypted for compliance. Currently the customer-service provides endpoints to create a customer (POST /customers), search or list customers (GET /customers?q=…), fetch a single customer (GET /customers/{id}), and GDPR-compliant deletion/anonymization (POST /customers/{id}/gdpr/delete) as well as data export[[5]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/customer-service/src/main.rs#L188-L196). **Similar to user accounts, customer records lack an update endpoint** for routine profile edits (e.g. correcting a typo in name or updating contact info), and there’s no general “deactivate” for customers apart from the permanent GDPR delete. Also, customers do not authenticate to the POS in the current design (they are associated with orders for loyalty but do not log into the system), so there is no password concept for customers yet. This may change if NovaPOS later provides a customer-facing app or portal, but in the current state, “customer accounts” function more as profile records.

**Admin Portal (React) – Current UI for User Management:** The Admin Portal includes a Users page, but its functionality is minimal. On page load it fetches the list of users for the current tenant and displays their name, email, and role[[6]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L461-L469). It also offers a form to invite a new user by entering name, email, role, and a temporary password[[7]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L262-L271)[[8]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L368-L376). When submitted, it calls the auth-service POST /users endpoint to create the account[[7]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L262-L271). The roles list is fetched from the backend or falls back to a static set of roles (super\_admin, admin, manager, cashier)[[9]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L105-L114)[[10]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L116-L124). **Crucially, the UI does not support any management actions on existing users** – there are no edit buttons, no way to change a user’s role or name, no ability to deactivate an account, and no mechanism to reset a user’s password. The Users page essentially stops at user creation[[11]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L66-L73)[[12]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L73-L79). If an admin needs to update or remove a user, there is no UI for it (and as noted, no API either). Additionally, **no audit information is shown** – the portal doesn’t display any history of user account events (creations, changes, logins, etc.), whereas it does have an audit log view for product changes in the Product management page. Internal review confirms that “user management stops at single user creation; there are no edit, deactivate, password reset, or audit flows for existing accounts”[[12]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L73-L79). The Admin Portal also currently lacks a dedicated interface to manage **customers** (e.g. viewing or editing customer profiles) – customer records are mainly handled behind the scenes via the POS and loyalty systems. This is another potential gap, since one can imagine admins wanting to look up a customer’s profile or merge duplicates, etc.

**Security and Multi-Tenancy:** Both the backend and frontend are designed for a multi-tenant context. Every user account (employee or customer) is tied to a tenant\_id, and the backend requires an X-Tenant-ID header on requests to scope data access[[3]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L393-L401). The Admin Portal ensures it sends this header (along with the user’s JWT token) for all API calls[[13]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L93-L101). For example, when an admin of tenant A fetches users, the request includes X-Tenant-ID: <tenant A id> and the auth-service query filters WHERE tenant\_id = $1 to return only users of that tenant[[3]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L393-L401). This prevents cross-tenant data leakage. A NovaPOS *super\_admin* (platform-wide admin) can manage multiple tenants – in the UI, a super admin gets a tenant dropdown to switch context and the selected tenant ID is sent in requests[[14]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L320-L329)[[15]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L76-L83). One current shortcoming is that the frontend doesn’t yet enforce role-based hiding of admin sections – any logged-in user could theoretically navigate to the Users page UI (though actual API calls might fail if they lack permission). There are no client-side route guards implemented yet[[12]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L73-L79). Strengthening this is part of the plan, ensuring that, for instance, only tenant admins (and super\_admins) see the “User Management” section, and only *super\_admin* sees cross-tenant features[[16]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L441-L449). The backend auth-service is expected to enforce roles as well (e.g. only admins can create users), though currently that enforcement may be rudimentary (it trusts that callers are authorized). Overall, the architecture provides a solid foundation (with JWT-based auth and tenant scoping), but **the feature set for user lifecycle management is incomplete**, and some security measures (RBAC enforcement, audit trails) need improvement.

## Gaps in User Management and Admin Tools

To summarize the key gaps before proposing solutions:

* **No User Update/Editing:** Once created, user accounts cannot be modified via API or UI. If an employee’s name changes, or if their role needs to be elevated/demoted, admins have no formal way to update those fields. This is highlighted in design docs as an acknowledged gap[[12]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L73-L79). A partial workaround might be deleting and re-creating the user, but that is not implemented either and would sever audit continuity.
* **No Account Deactivation/Activation:** There is currently no “disable user” function (no is\_active flag or equivalent). This means if an employee leaves the company or should be suspended, the only recourse is to possibly change their password or manually monitor that they don’t log in. An *is\_active* mechanism is needed for graceful deactivation without deletion. The current system’s absence of such a flag is noted, and adding it is considered (internal plan mentions possibly allowing disabling a user and implementing an endpoint for it)[[4]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L447-L455).
* **No Password Reset Workflow:** Users who forget their password cannot reset it (aside from an admin possibly creating a new account). The system has no “forgot password” email flow, and the admin portal provides no way for an admin to reset or set a new password for a user. This is a critical usability and security feature missing from MVP.
* **Limited Customer Account Management:** While not as prominently discussed as employee accounts, the ability to manage customer records (updating contact info, merging or deactivating accounts, etc.) is very limited. There’s a GDPR delete endpoint, but no standard update or soft-delete. Also, if in the future customers have login credentials for, say, an e-commerce portal, similar reset/deactivation capabilities would be needed. Currently, the admin UI has no section for customers, meaning any customer data fixes must be done via back-end tools or not at all.
* **Lack of Admin Audit Trail:** The admin portal does not surface any audit logs for user management actions[[17]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L76-L78). This means admins cannot easily track who created a user or changed a role, and there’s no visual log of security-related events (like account lockouts or password resets). On the backend, while some events (like MFA enrollment, login attempts) are logged or emitted to Kafka for monitoring, there isn’t a unified audit logging for administrative actions on user or customer records. Implementing such logging was identified as necessary to achieve a “world-class” ops experience (e.g., capturing user creation, updates, deletions)[[18]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L457-L461).

With these gaps identified, we propose a comprehensive implementation plan to enhance NovaPOS’s user management for both employees and customers, covering backend API improvements, frontend UX changes, multi-tenant security enforcement, and testing.

## Backend Enhancements: User Lifecycle APIs & Audit Logging

### 1. Add API Endpoints for User Updates and Deactivation

We will extend the **auth-service** API to support full user lifecycle operations beyond creation. The new endpoints will allow **updating user details**, **disabling/enabling accounts**, and **resetting passwords**. All new routes will be implemented with appropriate authentication (requiring a valid JWT) and authorization (requiring an admin role). For example, we will introduce a PUT /users/{id} route in auth-service to handle updates:

// In auth-service/src/main.rs, add the route for updating a user:  
.route("/users/:id", axum::routing::put(update\_user))

This update\_user handler will accept a JSON payload (e.g. { name?: String, role?: String, is\_active?: bool }) and the user ID as a path param. It will perform server-side validation and then execute an SQL update. **Only allowed fields will be updated** – e.g. name and role can be changed by admins. Changing email might be allowed too (with uniqueness check per tenant), though email changes could affect login and might be restricted. If a user’s role is updated, we will validate the new role against allowed values (using the same ALLOWED\_ROLES list the service already uses). If is\_active is present, the request is effectively toggling the user’s active status (see “deactivation” below). An example pseudo-code for the handler:

pub async fn update\_user(  
 State(state): State<AppState>,  
 auth: AuthContext, // Extracted JWT claims (for auth & role)  
 Path(user\_id): Path<Uuid>,  
 Json(update): Json<UpdateUserRequest>  
) -> Result<Json<User>, AuthError> {  
 // Ensure the caller has admin privileges  
 if !auth.has\_role("admin") && !auth.has\_role("super\_admin") {  
 return Err(AuthError::forbidden("Only admins can update users"));  
 }  
 // Enforce tenant scoping: ensure the target user belongs to the same tenant (unless super\_admin).  
 let tenant\_id = auth.tenant();   
 let user\_row = sqlx::query!("SELECT tenant\_id FROM users WHERE id = $1", user\_id)  
 .fetch\_optional(&state.db).await?;  
 if let Some(row) = user\_row {  
 if row.tenant\_id != tenant\_id && !auth.has\_role("super\_admin") {  
 return Err(AuthError::forbidden("Cannot modify users from another tenant"));  
 }  
 } else {  
 return Err(AuthError::not\_found("User not found"));  
 }  
 // Build SET clause dynamically based on provided fields  
 if let Some(new\_role) = update.role {  
 validate\_role(&new\_role)?;  
 }  
 // ... (similar checks for name format, etc.)  
 let result = sqlx::query\_as::<\_, User>(  
 "UPDATE users   
 SET name = COALESCE($1, name),   
 role = COALESCE($2, role),  
 is\_active = COALESCE($3, is\_active)  
 WHERE id = $4  
 RETURNING id, tenant\_id, name, email, role"  
 )  
 .bind(update.name)  
 .bind(update.role)  
 .bind(update.is\_active)  
 .bind(user\_id)  
 .fetch\_one(&state.db).await;  
 // Handle result...  
}

In the above, we use AuthContext (from common\_auth) to retrieve the caller’s roles and tenant. The auth-service JWT already encodes the user’s tenant and roles; using this, we **ensure only appropriate admins can update**. Tenant admins can only modify users within their tenant, while a super\_admin could update anyone (and even specify a different tenant in the request header if managing another tenant’s users). The code enforces that by comparing the target user’s tenant\_id in DB with the caller’s tenant, unless the caller is super\_admin. We also validate the role change against allowed roles to avoid introducing unknown roles[[19]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L29-L37). The SQL uses COALESCE for partial updates – this is one approach, or we could build the query dynamically. On success, it returns the updated user data. We will include the is\_active status in the returned User struct (after adding is\_active: bool to the User model in Rust and in the SELECT clause).

**Introduce is\_active flag:** A schema migration will add a boolean is\_active column to the users table (default true). This flag will indicate if a user account is active (allowed to log in) or deactivated. When is\_active is false, the **login endpoint will be modified** to refuse authentication (e.g., return an UNAUTHORIZED or FORBIDDEN error indicating the account is disabled). We will update the SQL query in login\_user to include a check like ...FROM users WHERE email=$1 AND tenant\_id=$2 AND is\_active = TRUE – ensuring inactive users are treated as if they don’t have valid credentials. Similarly, the list\_users query may either filter out inactive users or (better) include them and let the frontend display their status; we’ll opt to **include them** for visibility, but mark them in the response. The User JSON response will gain an is\_active field so the UI can, for example, gray out or label inactive accounts.

For **deactivation/reactivation** specifically, admins have two options in the API: (a) use the PUT /users/{id} with is\_active: false or true to change status, or (b) we could also provide convenience endpoints like POST /users/{id}/deactivate and /users/{id}/activate for clarity. Internally those would just set the flag accordingly. Given a RESTful design, we might stick with the single update endpoint and treat status as another editable field (with appropriate audit logging, see below). Under the hood, deactivation triggers a few things: besides setting is\_active=false in DB, we should **invalidate the user’s active sessions/tokens**. We have a refresh token store (auth-service likely tracks refresh tokens in DB), so upon deactivation we will delete any refresh tokens for that user (so they cannot obtain new access tokens). We will also design the front-end to prompt for re-login if an inactive user somehow stays logged in.

Importantly, we **do not delete user rows** when deactivating; we retain them for audit trail and potential reactivation. Reactivating simply sets is\_active=true again and allows the user to log in (the user might need a password reset if they forgot it during inactivity). This soft approach aligns with best practices, preserving historical records (e.g., who this user was associated with on past audits or orders).

### 2. Implement Password Reset Functionality

Supporting password resets is two-fold: allowing **users to recover access** if they forget their password, and enabling **administrators to trigger or assist in a reset** (especially if users are employees who might call IT for help). We will implement both an **admin-initiated reset API** and lay groundwork for a **self-service reset flow**.

**Admin-Initiated Reset:** A new endpoint, e.g. POST /users/{id}/reset-password, will allow an admin (or the user themselves, if logged in and allowed) to initiate a reset for the specified user account. We have a couple of design options:

* *Option A: Admin sets a temporary password directly.* In this approach, the admin provides a new password (or the system generates one) which is immediately applied to the user’s account. For example, the request could include {"temp\_password": "NewPass123!"} which the handler will hash and update in the database. The user can then use this password to log in (and ideally be prompted to change it upon first use). This is straightforward but has security downsides (the admin knows the user’s password). If we go this route, we will enforce that the provided temp password is strong and log that a password was reset for auditing.
* *Option B: Send a password reset email/link.* This aligns with typical “forgot password” flows. Here, calling POST /users/{id}/reset-password would generate a one-time **reset token** (a secure random string, stored in a new password\_reset\_tokens table with an expiry timestamp and associated user). The service would then send an email to the user’s email address with a password reset link containing that token. We would need to implement an outgoing email capability or integrate with a notification service. NovaPOS doesn’t currently have email delivery in place, but we could integrate a service (e.g., SMTP or SendGrid/SES via an async task or an event to a “notifications” microservice). The user would click the link, which would point to a **Password Reset UI** (we might create a simple page in the Admin Portal or a separate small web page) where they can enter a new password. That page would call a new endpoint like POST /users/reset-password/confirm with the token and new password, which the auth-service uses to validate the token and update the password.

For MVP simplicity, we might implement **Option A** initially (quickly allow an admin to set a new password) and plan for Option B (email-based flow) as a near-term enhancement for better security. In either case, the auth-service will use Argon2 to hash the new password (just as in user creation) and overwrite the password\_hash field[[2]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L363-L371). We will also increment or update any fields as needed (for instance, we might invalidate existing sessions similar to deactivation for security). The endpoint will require admin privileges (so a malicious user cannot reset someone else’s password).

Pseudo-code for a simple admin-driven reset might look like:

pub async fn admin\_reset\_password(  
 State(state): State<AppState>,  
 auth: AuthContext,  
 Path(user\_id): Path<Uuid>,  
 Json(req): Json<ResetPasswordRequest>  
) -> Result<Json<serde\_json::Value>, AuthError> {  
 if !auth.has\_role("admin") && !auth.has\_role("super\_admin") {  
 return Err(AuthError::forbidden("Only admins can reset passwords"));  
 }  
 // Only allow resetting within same tenant unless super\_admin:  
 // (similar tenant check as update\_user above)  
 // Generate new password if not provided (optional)  
 let new\_pass = req.new\_password.unwrap\_or\_else(generate\_temp\_password);  
 let new\_hash = hash\_password(&new\_pass).map\_err(|e| AuthError::internal\_error("Hashing failed"))?;  
 sqlx::query!("UPDATE users SET password\_hash = $1, failed\_attempts = 0, locked\_until = NULL WHERE id = $2",  
 new\_hash, user\_id)  
 .execute(&state.db).await?;  
 // Optionally, return the temp password so the admin can convey it to the user:  
 Ok(Json(json!({ "message": "Password reset successful", "tempPassword": new\_pass })))  
}

In this snippet, generate\_temp\_password could create a strong random password (e.g., 12 characters alphanumeric with symbols) if the admin didn’t supply one. We also reset the failed\_attempts and locked\_until to ensure the user isn’t still locked out from prior attempts. If we go with emailing a link instead, the logic would differ (generate token, store it, send email, and not immediately change the password), but given time constraints, the direct set approach can be delivered first. We will clearly audit these resets (see Audit Logging below), since a manual password reset is a sensitive event.

**Self-Service Reset:** In parallel, we can implement a POST /forgot-password endpoint (not necessarily part of admin portal, but part of auth-service’s public routes) where a user provides their email (and possibly tenant identifier) and if a matching account exists, the service generates a reset token and sends an email. This overlaps with Option B above. This likely requires setting up an email delivery mechanism (perhaps piggy-backing on an existing Kafka “security.alerts” topic or a new small mailer service). Even if not immediately used by the Admin Portal, having this API is important for user experience in the POS login screen (so an employee can reset their own password without involving an admin). We’ll ensure to implement it with rate-limiting (to prevent abuse) and token expiry (~1 hour). The confirmation endpoint to set a new password with a valid token will enforce strong password rules.

### 3. Extend Customer Service for Profile Updates (Customers)

For **customer accounts**, we will extend the customer-service to allow updating customer info and possibly deactivating customers in a softer way. Specifically, we plan to add a PUT /customers/{id} to allow modifications of a customer’s name, email, or phone number. This will be useful for admin users to correct typos or update contact information if a customer provides new details. The implementation will decrypt the existing fields as needed (customer-service currently stores email/phone encrypted[[20]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/customer-service/src/main.rs#L80-L88)), then re-encrypt any updated fields. Role-based access will be similar to other customer endpoints: all employees (cashier and up) might be allowed to update basic info[[21]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/customer-service/src/main.rs#L30-L38), but we could restrict sensitive changes (like an admin might be required to merge or delete accounts). The NewCustomer struct already exists, so we can reuse a similar struct for updates.

In terms of **deactivation**: Since customers don’t log into NovaPOS, “deactivating” a customer is less about access and more about status (e.g., marking a loyalty account as inactive or do-not-contact). We can introduce a boolean is\_active or a status field for customers if needed, but another approach is that the existing GDPR deletion covers the most extreme case (removing a customer entirely). For completeness, we may add is\_active to customers too, to allow marking a customer record as inactive without anonymizing it (for example, if a customer opts out of the loyalty program but we keep their purchase history). This would mirror the employee user approach. In that case, search and listing queries would either filter out inactive customers by default or label them. However, this is a lower priority compared to employee deactivation, and we could defer it unless a strong use case arises.

If we implement customer is\_active, we’d also add endpoints like POST /customers/{id}/deactivate or utilize the PUT to set the flag. The admin portal could then hide or mark inactive customers similarly. But again, since customers aren’t logging in, the primary need is likely editing and deleting. We will ensure the **GDPR delete functionality is accessible** via the admin UI (see frontend plan), since that’s the current method to “remove” a customer (it replaces their data with “[deleted]” per the implementation[[21]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/customer-service/src/main.rs#L30-L38)[[22]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/customer-service/src/main.rs#L73-L81)).

### 4. Audit Logging for User and Customer Lifecycle Events

To address the lack of audit trail, we will implement robust **audit logging** for all user management actions (and similarly for customer actions). This involves two parts: **capturing events in the backend whenever a lifecycle action occurs**, and **providing a way to retrieve and view those events**.

**Event Capture:** For each relevant function in auth-service, we will emit an event or log entry. Specifically: when a user is created, updated, deactivated/reactivated, or password reset, the system will record an audit event containing details like who performed the action (the admin’s user ID), timestamp, target user ID, and a description of the change. We have a couple of implementation approaches:

* *Via Kafka (or another event bus):* The auth-service already has Kafka integration for security events (e.g., MFA activities)[[23]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L23-L31)[[24]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L345-L353). We can define a new topic (or reuse an existing security/audit topic) and produce messages like user.created, user.updated, user.disabled, user.password\_reset etc. The message payload would include tenant\_id, target user\_id, actor user\_id, and any relevant metadata (e.g., changed fields). If a dedicated **Audit Service** exists or is planned, it could consume these events and store them centrally. The internal plan suggests sending events to an audit service if available[[25]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L457-L464). At minimum, writing these events to Kafka means we have a record for later analysis or monitoring (even if not immediately shown in the UI).
* *Via an Audit Log Table:* For a simpler MVP solution, we can write directly to a database table within auth-service (or a shared audit DB) whenever an action happens. For example, an audit\_log table with columns: id, tenant\_id, user\_id, action\_type, actor\_user\_id, timestamp, detail. Each row could capture one event. For instance, when an admin (actor 123) deactivates user (target 456), we insert a row: (uuid, tenant=XYZ, user\_id=456, action='deactivated', actor=123, time=now, detail='is\_active false'). This is easier to query for the UI later (no need to aggregate from Kafka). We might implement this in parallel to Kafka events for redundancy.

We will pursue **both**: produce a Kafka event (to integrate with any enterprise monitoring or future audit microservice) and insert into a local audit table for quick retrieval in the admin portal. The performance overhead is low given these events are infrequent and small.

**Audit Retrieval API:** To power the “visual audit trail” in the Admin UI, we will add endpoints to fetch audit logs. A straightforward addition is a read-only endpoint like GET /users/{id}/audit in auth-service, which returns the list of audit events for that user (sorted by time, most recent first). This will query the audit\_log table for entries matching that user\_id (and, importantly, matching the tenant of the requesting admin to prevent leaking cross-tenant info). Alternatively, we could offer a more general audit query by tenant or by various filters, but per-user is a focused need for now. The response might look like:

[  
 {  
 "timestamp": "2025-09-29T12:00:00Z",  
 "action": "created",  
 "actor": "Alice (admin@example.com)",  
 "details": "User created with role Manager"  
 },  
 {  
 "timestamp": "2025-11-01T09:30:15Z",  
 "action": "password\_reset",  
 "actor": "Bob (sysadmin@example.com)",  
 "details": "Password reset (temp password issued)"  
 },  
 {  
 "timestamp": "2025-11-01T09:31:00Z",  
 "action": "deactivated",  
 "actor": "Bob (sysadmin@example.com)",  
 "details": "Account disabled"  
 }  
]

(Where actor could be enriched with name/email for convenience, or just an ID that the UI can resolve to a name by cross-referencing the users list.) Each entry’s details field might include extra info like what fields changed (e.g., “role changed from Cashier to Manager”), though we should be cautious not to log sensitive data. For a password reset, for example, we would **not** log the actual temp password, just that a reset occurred.

The auth-service can enforce that only authorized roles fetch audit logs. Likely, any admin of the tenant can view audit entries for users of that tenant. We might restrict managers from seeing audit logs if we consider that sensitive, but since they can’t perform these actions anyway, it may be fine to let only admins/super\_admins access audit trail endpoints.

For **customer-service**, we will implement a similar audit mechanism: logging events such as customer.created, customer.updated, customer.deleted (GDPR deletion). Since customer-service already deals with PII and encryption, we’ll ensure not to log raw sensitive data. But an audit entry might note that “Admin X updated customer Y’s email (old vs new hashed values)” or simply “updated profile” for brevity. We can store these in a customer\_audit table or send events to the same audit topic.

By implementing audit logging, we not only satisfy the need for a UI-visible trail[[17]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L76-L78) but also bolster security – any critical action on accounts will be trackable. This addresses an internal risk that previously “user events have no surfaced history”[[17]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L76-L78). Our approach ensures that moving forward, **all user lifecycle changes are recorded** for compliance and debugging purposes.

### 5. Hardening Security and Multi-Tenant Integrity

In addition to the new functionality, we will take this opportunity to **enforce security best practices** in the user management APIs:

* **Role-Based Access Control (RBAC):** We will explicitly enforce that only the proper roles can call the new endpoints. For auth-service, this means only users with role admin or super\_admin (and possibly manager for limited actions if we decide, but likely not) can hit create/update/deactivate/reset routes. Currently, the system relies on the UI to limit these actions, but we will add server-side checks using the JWT claims (AuthContext). If an unauthorized role calls these endpoints, the service will return HTTP 403 Forbidden. This closes the gap where “mid-tier roles can reach privileged forms” by simply guarding on the backend as well[[12]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L73-L79).
* **Tenant Isolation Checks:** As demonstrated in the pseudo-code, every operation will verify that the targeted resource belongs to the same tenant as the requesting admin (unless the admin is a super\_admin managing another tenant’s data intentionally). This prevents any tampering with URL paths or IDs to manage accounts across tenants. The X-Tenant-ID header and JWT tenant claim act as a double safeguard. We will continue the pattern used in list queries[[3]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L393-L401) for all new queries and updates, e.g., adding AND tenant\_id = $tenant\_id conditions in SQL updates as well.
* **Input Validation:** We’ll enforce strong validation on inputs – for instance, ensuring emails are well-formed (the UI already does a simple regex check[[26]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L243-L251), but we will do similar on backend), password strength checks (especially if admins set passwords manually), and checking that required fields are not empty. This prevents bad data from entering the system via the new APIs.
* **Preventing Dangerous Actions:** Certain actions like deleting a user outright won’t be provided at all (to avoid accidental loss of data; deactivation covers the need). If in the future a delete is needed, it might only be allowed for super\_admin on users (for example, removing test accounts), and even then we’d likely implement it as a flag rather than physical deletion.
* **Sessions and Tokens:** As noted, upon critical changes (like role change, deactivation, or password reset), we should consider invalidating active sessions for that user. We have a refresh token store (likely in auth\_refresh\_tokens table). We will implement logic to revoke tokens for a user when these events happen (except perhaps name or role change might not require it, but definitely for password reset or deactivation). This ensures, for example, if an account is disabled or password changed, the user cannot continue using an old token. The token signer in auth-service can be extended with a method to revoke a user’s refresh tokens (e.g., deleting them where user\_id matches). For immediate logout, we might also maintain a token blacklist or use token invalidation strategies, but given short JWT lifespans (15 minutes by default) and removal of refresh tokens, the window of risk is small.
* **Logging and Monitoring:** All new endpoints will log their activity (in addition to audit logs). For instance, the auth-service will info! log “Admin {actor\_email} deactivated user {target\_email} (id)” along with tenant context. These logs combined with audit events provide traceability. If we have a monitoring system, we could even raise alerts for certain sensitive actions (e.g., a spike in password resets or multiple deactivations could indicate an issue).

By implementing the above, we address multi-tenant security considerations: each tenant’s data remains walled off, only proper roles can manage accounts, and every change is tracked and reviewable. This meets the MVP security requirements and builds trust in the system’s admin controls[[27]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L471-L479).

## Frontend (Admin Portal) Enhancements: UX Design & Code Structure

On the frontend, the Admin Portal will be upgraded to expose the new user management capabilities through an intuitive UI. We will extend the **Users page** to allow administrators to perform the lifecycle actions (update info, deactivate/reactivate, reset password) and view audit history for each user. Additionally, we will introduce a new **Customers page** in the Admin Portal to manage customer accounts in a similar vein.

### 1. Users Management UI – Edit, Deactivate, Reset, Audit

**UI Layout:** We will augment the existing **UsersPage** component, which currently shows “Add New User” and the list of current users[[28]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L432-L441)[[29]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L444-L452). The following changes are planned:

* **Add an Actions column in the users table:** We will append a new <th>Actions</th> in the table header and render action buttons for each user row. For example, for each user in sortedUsers.map(...), we’ll include something like:

<td className="px-4 py-2 text-sm">  
 {user.is\_active ? (  
 <button onClick={() => handleDeactivate(user)} className="text-red-600 hover:underline">Deactivate</button>  
 ) : (  
 <button onClick={() => handleActivate(user)} className="text-green-600 hover:underline">Activate</button>  
 )}  
 <button onClick={() => openEditModal(user)} className="ml-4 text-blue-600 hover:underline">Edit</button>  
 <button onClick={() => handleResetPassword(user)} className="ml-4 text-indigo-600 hover:underline">Reset&nbsp;PW</button>  
</td>

These action buttons let an admin deactivate/activate a user, edit their details, or reset their password. We use simple text buttons with distinctive colors (red for deactivate, etc.) or icons if we have an icon set. On small screens, we might collapse these into a dropdown “More” menu to save space – but as MVP, a horizontal list of actions is acceptable.

* **Edit User Modal/Dialog:** Clicking “Edit” opens a modal dialog (or drawer) with a form to edit the user’s information. This form will be similar to the “Add New User” form, but without the password field (since we are not changing password here) and with the fields pre-filled. The admin can update the user’s **name** and **role**. (Email change could be allowed, but we should caution that changing the login email might confuse the user; we can decide to allow it if needed.) We will include an Active toggle in this form as well, so the admin can also deactivate/reactivate here. For example:

**Edit User Modal Design:**  
- Fields: Name (text input), Email (read-only or editable based on decision), Role (dropdown of roles), Active (checkbox or toggle).  
- Buttons: “Save Changes” and “Cancel”.

The modal will use component state to hold changes; on submit it calls the new PUT /users/{id} API. We’ll create a hook or function updateUser in our API utilities to send this request. After a successful update, we will refresh the user list (or optimistically update the state for that user in users array). If the current user being edited is the one performing the action and they changed their own role or email, we might want to prompt them to re-login or update the current session data – but since typically an admin edits others, we can skip that complexity.

* **Deactivate/Activate:** The “Deactivate” button will trigger a confirmation (we should confirm destructive actions). A simple window.confirm("Deactivate this user?") might suffice for MVP, or a nicer modal “Are you sure? This user will no longer be able to log in.” On confirmation, we call the update API with is\_active: false. The handleDeactivate function will call something like:

await fetch(`${AUTH\_SERVICE\_URL}/users/${user.id}`, {  
 method: 'PUT',  
 headers: buildHeaders(true),  
 body: JSON.stringify({ is\_active: false })  
});

If successful, it will update that user’s entry in the users state (or refetch the list via our existing fetchUsers function for simplicity). The UI will then update to show that user as inactive (perhaps we gray out their row text and the button now says “Activate” as implemented above). Similarly, the “Activate” button does the inverse.

We will visually distinguish inactive users in the list: possibly by adding a CSS class to the row to make the text lighter or adding a badge “Inactive” next to their name. This helps admins immediately see who is disabled. We might also move inactive users to the bottom of the list or provide a filter toggle (not necessary for MVP if the list is short, but useful as the user base grows). For now, a simple indicator is fine.

* **Reset Password:** Clicking “Reset PW” will initiate the password reset flow. We have a couple of UI approaches corresponding to backend Option A or B:
* If using **admin-set password** (Option A), clicking reset could open a modal prompting the admin to enter a new temporary password for the user (with validation for strength). The modal would have a field “New Temporary Password” (or a generate button if we want to auto-generate). Upon submit, it calls POST /users/{id}/reset-password with the chosen password. The backend responds (possibly echoing the temp password if generated). We then display a confirmation: “Password has been reset. Communicate the new password to the user securely.” If the password was system-generated and returned, we show it in a copyable text box for the admin.
* If using **email link** (Option B), clicking reset can immediately call the API which triggers an email, and then we show a message: “Password reset link sent to the user’s email.” We might still confirm with the admin before sending (to avoid spamming by mis-click).

For MVP, a straightforward flow could be: click “Reset PW” -> confirm dialog “Reset password for this user? They will need to use the new password sent to them.” -> on OK, call API -> on success, show an alert/notification. We will implement a state for successMessage or similar (as seen already in UsersPage for user created message[[30]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L270-L279)) to show feedback.

* **Audit Trail View:** To give a “visual audit trail per user,” we will implement a **User Detail drawer or page** that shows the user’s info and a list of recent actions. One design is to allow clicking on a user’s name in the table as a hyperlink – when clicked, it opens a side panel or navigates to a UserDetailsPage. Given our routing structure, adding a new page might be cleaner. For example, add a route: <Route path="/users/:userId" element={<UserDetailsPage />} />. On the UsersPage, wrap the name or entire row in a Link to /users/${user.id} for admins.

The **UserDetailsPage** will fetch the full user object (we might already have it from the list, but to be safe or to get fresh data including perhaps last login time if we ever provide that) and call the new audit log endpoint GET /users/{id}/audit. We will display the user’s profile info at top (name, email, role, status), and below that, a chronologically ordered list of audit events. Each event can be shown in a timeline or simple list format:

[Nov 1, 2025 09:31] \*\*Deactivated\*\* by Bob (SysAdmin)  
[Nov 1, 2025 09:30] \*\*Password Reset\*\* by Bob (SysAdmin)  
[Mar 12, 2025 16:05] \*\*Role changed\*\* from Cashier to Manager by Alice (Admin)  
[Jan 5, 2025 09:12] \*\*Account created\*\* by Alice (Admin)

We can style it with timestamps and bold action keywords, as in the example. This gives a quick audit trail. We’ll likely use a state like userEvents to store the fetched array. We should handle the case of no events (just show “No history available” if none, though create event at least should be there). We will also include a “Back to Users” link/button to return to the main list.

Optionally, we could integrate this detail view as a slide-out panel on the Users page to avoid full page navigation. A slide-out (drawer) could be triggered by an “View Details” action. However, to keep it simple, a separate page is fine and easier to implement with routing.

* **Role-Based UI Restrictions:** We will fix the current flaw where non-admins could see the Users page. Using the AuthContext (currentUser.role), we’ll redirect or hide content if the role is insufficient. For example, in UsersPage’s effect, if currentUser.role is not admin or super\_admin, we will navigate('/home') or show a “Not Authorized” message. Additionally, in the main navigation menu (likely in a layout or App component), we will conditionally display the “Users” link only for appropriate roles[[31]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L439-L446)[[16]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L441-L449). This can be done with something like:

{currentUser?.role === 'admin' || currentUser?.role === 'super\_admin' ? (  
 <NavLink to="/users">User Management</NavLink>  
) : null}

Similarly for any future “Integrations” or sensitive pages. This ensures that, e.g., a manager who logs into Admin Portal doesn’t even see the option to manage users. It’s a usability improvement and reduces temptation/trial-and-error. (The backend will also reject unauthorized calls, but this is about the UI experience.)

**State Management and Code Structure:** The UsersPage is currently a single component handling create and list. With the new features, the code will grow. To maintain clarity, we may refactor a bit:

* Extract sub-components for clarity. For instance, create UserEditModal as a separate component to encapsulate the modal logic, or a UserRowActions component purely for rendering the action buttons and handling their onClick events. This can keep the UsersPage from becoming too monolithic.
* Introduce custom hooks for data operations: We already have inline fetchUsers, fetchRoles, etc. We might refactor these into hooks like useUsers(tenantId) that returns the list and loading state, or a hook useUserAudit(userId) for fetching audit logs in the detail page. This is optional but would align with a clean React pattern.
* Ensure reusability of API call logic: possibly have an api.ts or similar in the frontend where functions like updateUser(id, data), resetUserPassword(id), etc., are defined, to avoid duplicating fetch call code across components. Given the small size, inline is fine, but anticipating growth, an abstraction helps.

**Multitenancy in UI:** Since super\_admin can select a tenant to view/manage, our UsersPage already handles a tenant switcher (the dropdown for super\_admin to pick a tenant, implemented in fetchTenants and state like selectedTenantId)[[32]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L76-L84)[[33]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L146-L154). We will ensure all new actions respect the selected tenant context. For example, when a super\_admin triggers an action on a user, the buildHeaders function already can take an override tenant ID[[34]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L94-L101); we will use it so that the request carries the correct X-Tenant-ID (the code already uses effectiveTenantId for listing and creation). Similarly, in UserDetailsPage, if a super\_admin is viewing a user of another tenant, we should include that tenant header on the audit fetch call. We might pass the tenantId as part of Link state or simply rely on the user’s tenant\_id (which is in the user object) and include that in the header. In essence, the front-end will continue to enforce that an admin only manages one tenant at a time, with super\_admin able to switch context.

**UI for Customer Management:** While the core question focuses on user (employee) accounts, we will also build a **CustomersPage** in the admin portal to handle customer accounts. This page will allow admins/managers to search for customers and view/edit their details, ensuring a unified admin experience. Key elements:

* A search bar to lookup customers by name, email, or phone (utilizing the GET /customers?q= endpoint). We can implement a simple controlled input and call the API when the user submits or after a debounce.
* A results list/table of customers similar to the Users table (Name, Email, Phone, maybe Created Date). On selecting a customer, either open a detail view or inline expand to show more info.
* An edit function: allow updating name, email, phone. Likely as a modal similar to user edit. This calls the new PUT /customers/{id} we implement. We’ll handle encryption on backend, so from UI perspective it’s just sending plaintext (the backend API may accept it normally and do encryption server-side).
* If we want to allow GDPR delete from UI, we could put a “Delete” button in the customer detail view. Clicking it would warn “This will permanently anonymize the customer and cannot be undone” and then call POST /customers/{id}/gdpr/delete. After that, the UI could refresh or remove that customer from the list.
* Audit trail for customers: If we implement audit logging for customers, we could also show a history of changes for a customer (e.g., profile updated, or GDPR delete performed by X). This could be another detail panel. However, customer audit might be less critical for MVP unless actively managing lots of customer data.

The CustomersPage would likely be visible to any role that can view customers (maybe even cashiers to look up info, but editing/deleting perhaps only admin). We can start by restricting it to admin/manager roles if that fits business rules (managers often handle customer issues). This can be configured similarly via role checks.

From a **code scaffolding** perspective, we might structure it as follows in frontends/admin-portal/src/pages/CustomersPage.tsx (new file):

* State: customers list, query string, loading, error, maybe selected customer for detail modal.
* useEffect or event handler to fetch customers on search.
* Render: a search input at top; below, either a table of results or if no query, perhaps a message “Search for a customer by name, email, or phone.” Each result row could have an “Edit” or “View” action.
* Possibly reuse some styles and components from UsersPage (we can generalize an AdminSection layout, etc.).

In terms of navigation, add a route in App.tsx: <Route path="/customers" element={<CustomersPage />} /> and a corresponding nav link “Customers” (visible if the user’s role is at least manager perhaps). This fills a gap in the current portal (customer management was absent).

### 2. UX Considerations for Smooth Admin Workflows

The UX flows will be designed to be intuitive:

* After creating a new user, the form already clears and a success message is shown[[30]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L270-L279). We will maintain that behavior, and perhaps also automatically show the new user in the list (the current code already refreshes the list after creation[[35]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L270-L278)).
* When editing a user or performing actions, we’ll provide timely feedback. E.g., after saving edits in the modal, we can show a toast or message “User updated successfully” (the UsersPage already has a mechanism for successMessage, which we can reuse for various actions). For deactivate/reactivate, a small confirmation message like “User deactivated” can appear.
* We will handle error cases gracefully: if an update fails (network or server error), show an error banner (similar to how creation errors are shown[[36]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L277-L284)). If a password reset fails (e.g., email not sent or no permission), inform the admin.
* For audit trail, ensure it updates in near-real-time – e.g., if an admin deactivates a user and then opens that user’s detail, the deactivation event should appear. Since we’ll log the event server-side before returning success, by the time the admin opens the audit panel the event will be present. If needed, we could even update an in-memory audit list if the admin is viewing that user detail as they make changes, but that scenario is rare (likely they trigger action from the list view). Simpler is to fetch fresh when opening the detail.
* **Responsiveness and UX**: Make sure modals and tables work on different screen sizes. If a modal is too large, make it full-screen on mobile. The table might need horizontal scrolling (we already have overflow-x-auto on it[[29]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L444-L452)). The action buttons on each row should degrade gracefully (maybe stack vertically on narrow screens). These are minor UI tweaks we can test.
* **Visual cues**: Use consistent icons or colors for certain actions (e.g., a lock icon for deactivate, an unlock for activate, a key icon for reset password). Even if we just use text, color-coding helps (red = destructive, etc.). We should also label the modals clearly, e.g., “Edit User” title on the modal, and in a reset password modal, warn about what will happen.
* **MFA and advanced security**: The Implementation Plan mentioned possibly prompting admins to enroll MFA devices[[16]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L441-L449). While our focus is user management, we should ensure that if the backend requires MFA for certain actions, the UI handles it. For instance, if an admin’s token is not MFA-verified and the backend returns MFA\_REQUIRED for a sensitive endpoint, the UI should prompt accordingly. This might be an edge case if require\_mfa is enabled for admin role. We may not implement it now, but keep the design open for adding an MFA confirmation step if needed (like a modal to enter an MFA code before, say, deleting a customer or similar high-security action).

### 3. Code Examples (Frontend)

Below are brief code scaffolds to illustrate parts of the frontend implementation:

**Adding actions to the Users table (excerpt):**

<thead>...<th>Actions</th></thead>  
<tbody>  
 {sortedUsers.map(user => (  
 <tr key={user.id} className={!user.is\_active ? 'opacity-50' : ''}>  
 <td>{user.name}</td>  
 <td>{user.email}</td>  
 <td>{roleLabel(user.role)}</td>  
 <td>  
 {user.is\_active ? (  
 <button onClick={() => onDeactivate(user)} className="text-red-600">Deactivate</button>  
 ) : (  
 <button onClick={() => onActivate(user)} className="text-green-600">Activate</button>  
 )}  
 <button onClick={() => openEdit(user)} className="ml-3 text-blue-600">Edit</button>  
 <button onClick={() => onResetPassword(user)} className="ml-3 text-indigo-600">Reset PW</button>  
 </td>  
 </tr>  
 ))}  
</tbody>

Here, we dim the row (opacity-50) if the user is inactive. The event handlers like onDeactivate will call the appropriate fetch API and then update state. For example:

const onDeactivate = async (user) => {  
 if (!window.confirm(`Deactivate user ${user.email}?`)) return;  
 try {  
 const res = await fetch(`${AUTH\_SERVICE\_URL}/users/${user.id}`, {  
 method: 'PUT',  
 headers: buildHeaders(true),  
 body: JSON.stringify({ is\_active: false })  
 });  
 if (!res.ok) throw new Error('Failed');  
 setSuccessMessage(`${user.email} has been deactivated.`);  
 // Refresh or update state:  
 setUsers(prev => prev.map(u => u.id === user.id ? { ...u, is\_active: false } : u));  
 } catch (err) {  
 setError("Failed to deactivate user. Please try again.");  
 }  
};

We’ll implement similar for onActivate. For onResetPassword, if using simple admin-set method, we might do:

const onResetPassword = async (user) => {  
 if (!window.confirm(`Send a password reset for ${user.email}?`)) return;  
 try {  
 const res = await fetch(`${AUTH\_SERVICE\_URL}/users/${user.id}/reset-password`, {  
 method: 'POST',  
 headers: buildHeaders(true),  
 body: JSON.stringify({ }) // possibly include new password if we want to specify  
 });  
 if (!res.ok) throw new Error('Failed');  
 const result = await res.json();  
 if (result.tempPassword) {  
 prompt(`Password reset successful. Temporary password:`, result.tempPassword);  
 }  
 setSuccessMessage(`Password reset initiated for ${user.email}.`);  
 } catch {  
 setError("Failed to reset password. Please try again.");  
 }  
};

(If the backend returns a temp password, we show it in a prompt or some UI for the admin; if it sends email, just a success message is enough.)

**User Edit Modal component (simplified):**

function UserEditModal({ user, onClose }) {  
 const [form, setForm] = useState({ name: user.name, email: user.email, role: user.role, is\_active: user.is\_active });  
 const [saving, setSaving] = useState(false);  
 const { token, currentUser } = useAuth();  
 const saveChanges = async () => {  
 setSaving(true);  
 try {  
 const res = await fetch(`${AUTH\_SERVICE\_URL}/users/${user.id}`, {  
 method: 'PUT',  
 headers: { ...buildHeaders(true), Authorization: `Bearer ${token}` },  
 body: JSON.stringify({   
 name: form.name,   
 role: form.role,   
 // email change if allowed  
 is\_active: form.is\_active   
 })  
 });  
 if (!res.ok) throw new Error();  
 const updated = await res.json();  
 onClose(updated); // pass updated user back to parent  
 } catch {  
 // handle error (could pass error state up or show here)  
 onClose(null);  
 } finally {  
 setSaving(false);  
 }  
 };  
 return (  
 <div className="modal">  
 <h2>Edit User</h2>  
 <label>Name: <input value={form.name} onChange={e => setForm({...form, name: e.target.value})}/></label>  
 <label>Role:   
 <select value={form.role} onChange={e => setForm({...form, role: e.target.value})}>  
 {roleOptions.map(r => <option key={r.value} value={r.value}>{r.label}</option>)}  
 </select>  
 </label>  
 <label>Active:   
 <input type="checkbox" checked={form.is\_active} onChange={e => setForm({...form, is\_active: e.target.checked})}/>  
 {form.is\_active ? 'Active' : 'Inactive'}  
 </label>  
 <button onClick={saveChanges} disabled={saving}>Save</button>  
 <button onClick={() => onClose(null)}>Cancel</button>  
 </div>  
 );  
}

When the modal closes, we update the user list state in UsersPage if an updated user object is returned. This pattern ensures the UI reflects changes immediately.

**Customers Page outline:**

const CustomersPage = () => {  
 const [query, setQuery] = useState('');  
 const [customers, setCustomers] = useState([]);  
 const [selectedCustomer, setSelectedCustomer] = useState(null);  
 const { token, currentUser } = useAuth();  
 const searchCustomers = async () => {  
 try {  
 const res = await fetch(`${CUSTOMER\_SERVICE\_URL}/customers?q=${encodeURIComponent(query)}`, {  
 headers: { Authorization: `Bearer ${token}`, 'X-Tenant-ID': currentUser.tenant\_id }  
 });  
 const results = await res.json();  
 setCustomers(results);  
 } catch (err) {  
 console.error('Customer search failed', err);  
 }  
 };  
 return (  
 <div className="admin-section">  
 <h2>Customers</h2>  
 <input type="text" value={query} placeholder="Search customers..."   
 onChange={e => setQuery(e.target.value)} />  
 <button onClick={searchCustomers}>Search</button>  
 <table>  
 <thead><tr><th>Name</th><th>Email</th><th>Phone</th><th>Actions</th></tr></thead>  
 <tbody>  
 {customers.map(c => (  
 <tr key={c.id}>  
 <td>{c.name}</td><td>{c.email ?? '—'}</td><td>{c.phone ?? '—'}</td>  
 <td>  
 <button onClick={() => setSelectedCustomer(c)}>View</button>  
 {/\* We can allow direct edit or delete here as well \*/}  
 </td>  
 </tr>  
 ))}  
 </tbody>  
 </table>  
 {selectedCustomer && <CustomerDetailModal customer={selectedCustomer} onClose={() => setSelectedCustomer(null)} />}  
 </div>  
 );  
};

The CustomerDetailModal would show full info and have an edit form similar to user edit (but simpler fields), and possibly a “Delete Customer” button that calls the GDPR delete endpoint. We will also show maybe a list of that customer’s recent orders or loyalty points if that data is easily accessible, to give context – though that’s beyond core “account management”, it’s a nice-to-have. At minimum, allow editing and deleting.

**Route integration:** We add in our App component’s routes:

<Route path="/users" element={<UsersPage />} />  
<Route path="/users/:id" element={<UserDetailsPage />} />  
<Route path="/customers" element={<CustomersPage />} />

and ensure the navigation menu has entries for “Users” (for admin) and “Customers” (for roles that manage customers, e.g. admin, manager).

By implementing these UI components, the Admin Portal will evolve from a **simple user creation form** to a full **User Management dashboard**, where all key account lifecycle tasks are possible. This directly addresses the earlier limitation that “user management stops at single user creation”[[12]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L73-L79). Now, admins will be empowered to manage access throughout the user’s lifecycle, and do so with visibility into historical changes.

## Multi-Tenancy & Security Considerations (Frontend & Backend)

Throughout the plan, multi-tenant support and security have been woven in, but to recap and add additional measures:

* **Isolation by Tenant:** All backend operations require a tenant context. The Auth and Customer services use the provided X-Tenant-ID header (and also validate it against the JWT’s tenant claim). We will maintain this in all new API calls from the frontend. In fact, our buildHeaders() utility in the Admin Portal already ensures the current or selected tenant ID is attached to every request[[34]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L94-L101). We must be careful when viewing cross-tenant data as a super\_admin: e.g., if super\_admin A is managing tenant B’s users, and they click “reset password” for a user in tenant B, the header should still be X-Tenant-ID: B (which our state logic covers by effectiveTenantId usage). We will test scenarios of tenant switching to ensure no leakage (for instance, ensure that if a super\_admin forgets to select a tenant, our code defaults to their own tenant or blocks actions accordingly).
* **Frontend Role-Based Access:** We touched on this, but essentially we will **prevent unauthorized UI access**. Not only hiding menu items, but also implementing route guards. One approach: define a higher-order component or a wrapper for protected routes that checks the role. For example:

const AdminRoute = ({ children }) => {  
 const { currentUser } = useAuth();  
 return (currentUser && ['admin','super\_admin'].includes(currentUser.role))   
 ? children   
 : <Navigate to="/home" replace />;  
};  
// Usage:  
<Route path="/users/\*" element={<AdminRoute><UsersPage/></AdminRoute>} />

And similarly for Customers if managers are allowed, etc. This ensures even if someone manually enters the URL, they get redirected if they’re not allowed. Coupled with backend checks, this secures the flow.

* **Permissions Matrix:** In multi-tenant SaaS, typically:
* *Cashier:* no access to admin portal (they use POS only).
* *Manager:* perhaps limited access (maybe can view customers or run reports, but not manage users).
* *Admin:* full tenant admin (manage users, products, view all data for their tenant).
* *Super Admin:* platform admin (manage tenants and everything).

Our implementation aligns with this: we’ll restrict “User Management” and “Settings (tenants, integration keys)” to admin/super\_admin only[[16]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L441-L449). We might allow “Customers” page for manager as well if desired (as they might handle customer issues), depending on requirements. This can be configured easily by adjusting the allowed roles check in our route or menu logic.

* **Cross-Tenant Admin Actions:** Only super\_admin can create new tenants and manage integration keys (the Settings page handles that[[11]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L66-L73)). That flows through auth-service as well. We will ensure our changes don’t inadvertently allow a tenant admin to act outside their scope. For example, the PUT /users/{id} endpoint will not allow a tenant admin of tenant X to update a user of tenant Y because of the tenant check logic we add. We will test that super\_admin (with an override header) *can* update across tenants (since that might be needed if they are helping a tenant’s admin or fixing an issue). The UI for super\_admin already allows selecting a tenant for user creation; we might also allow super\_admin on the Users page to view all tenants at once, but currently it’s one at a time via selection, which is fine.
* **Sensitive Data Protection:** We will continue to ensure that sensitive data (password hashes, reset tokens, etc.) never hits the frontend or user’s browser. Our API responses for user objects do not include password\_hash (the Rust User struct explicitly omits it[[37]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L145-L153)). When we implement password reset tokens, those will only be sent to emails or returned to an admin in hashed form if at all. The customer-service already hashes or encrypts PII before returning (for example, it likely doesn’t return raw phone/email unless decrypted – looking at Customer struct in code[[38]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/customer-service/src/main.rs#L56-L65), it returns decrypted email/phone as plain text in the API, which is fine because only authorized roles can call it). We should be mindful not to log or expose secrets. For instance, if an admin copies a temporary password out of the UI, they should do so over a secure channel when giving it to the user.
* **Compliance and Audit:** The audit logging improvements bolster compliance (knowing who did what). We should ensure audit logs themselves are protected – only admins of that tenant or super\_admin can view them. We wouldn’t want, say, a manager to see that an admin reset someone’s password if that’s deemed sensitive (though arguably it’s fine within the same tenant). Possibly limit audit view to admin roles.
* **Testing for Security:** We will incorporate tests (see next section) to simulate actions by different roles to verify that, e.g., a manager cannot call admin-only endpoints (should get 403), an admin cannot affect another tenant, etc.

By addressing these considerations, we maintain a **secure multi-tenant environment** even as we expand functionality. Each tenant’s data and user accounts are strictly separated, and only those with proper authority can manage accounts, reducing risk of privilege abuse or mistakes.

## Testing Plan: Ensuring Reliability and Security

To confidently launch these enhancements, we will write comprehensive tests for both backend and frontend:

### Backend Tests

**Unit Tests (Rust):** We will add unit tests in the auth-service for the new handlers logic:

* **Update User Tests:** Simulate various scenarios:
* Admin updating a user’s name/role within their tenant (expect success, verify DB changes).
* Admin attempting to update a user from another tenant (expect forbidden error).
* Manager (or lower role) calling update endpoint (expect forbidden).
* Super admin updating a user of a specified tenant (with appropriate header) – should succeed.
* Trying to set an invalid role (expect validation error).
* Ensuring that partial updates leave other fields unchanged.
* Concurrent update scenarios if possible (two updates at once) to ensure no deadlock or weird behavior (likely fine with Postgres row lock).
* **Deactivate/Activate Tests:**
* Admin deactivates a user and that user can no longer log in. We can simulate by: create user, deactivate them, then attempt a login via the login\_user function with their credentials (expect AuthError or no token issued).
* Reactivate and then login succeeds.
* Ensure that deactivating doesn’t delete the user and that they still appear in list\_users. Also verify that list\_users indeed returns both active and inactive (or adjust if we decide to filter).
* **Password Reset Tests:**
* For Option A (direct set): After resetting, verify the password\_hash in DB has changed and that the old password no longer works while the new password does. We can call login\_user with old vs new password.
* Ensure that an admin from a different tenant cannot reset someone else’s user (similar tenant check).
* If we do the token/email method, we would test token generation and validation logic (like, create token, ensure it’s valid, after use or expiry it’s invalid, etc.).
* **Audit Logging Tests:**
* Perform a simulated action (like call create\_user or update\_user in a test transaction) and then query the audit\_log table to ensure an entry was added with correct details.
* Test that for combined actions, multiple logs are recorded properly (e.g., update that changes role and active status – we might log one event or two; probably one event “updated” with detail including both changes).
* If Kafka events are used, we might mock or intercept the Kafka producer to assert a message was sent (though that might be more of an integration test).
* **Customer Service Tests:**
* Test the new PUT /customers/{id}: update fields and check DB reflects it (including verifying encryption by reading the encrypted column if possible).
* Unauthorized updates (e.g., cashier updating if we disallow, or cross-tenant).
* GDPR delete already likely has tests, but test UI integration if needed (like calling delete sets name = “[deleted]”, etc.).

We will run these tests against an ephemeral test database (the repository seems to use sqlx::query! with offline data, but we can still do integration-style tests with a local Postgres or an in-memory SQLite if supported). The cargo sqlx prepare offline data means we might need to regenerate that after adding queries, which is part of the dev workflow. We’ll include that as needed.

**Integration Tests (API level):** We can write tests using a framework like reqwest against a running instance of the services (maybe in a test mode). For example, spin up the auth-service (with a test DB), create a test admin user, then simulate HTTP calls to the new endpoints (this is more involved but can catch routing issues and auth). Alternatively, use something like Axum’s tower tests to call the routes directly with a mocked State and AuthContext. This ensures the wiring in main.rs (routes definitions[[1]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/main.rs#L134-L141)) matches what we expect.

**Security Tests:** We’ll include tests for permission boundaries: - Ensure an admin token with role=admin cannot create a super\_admin user (if we decide to restrict that – possibly only super\_admin can create another super\_admin to avoid privilege escalation). If needed, we’ll enforce that in create\_user (the config might already restrict certain roles). - Ensure that a user with no JWT (or invalid JWT) gets 401 on these endpoints (this might be handled by global auth middleware if in place, or we may need to explicitly check AuthContext presence). - Try some injection or malformed data in inputs to verify our validations (e.g., extremely long name or weird characters, which should be fine, or SQL injection attempts in fields – using parameterized queries prevents injection, but tests affirming that nothing breaks).

### Frontend Tests

We will use React Testing Library and/or Jest to test the frontend behavior:

* **Component Rendering Tests:** Ensure that the UsersPage renders the list of users and the action buttons when given sample data. We can mock the context to provide a currentUser (role admin) and stub out the fetch calls by mocking global.fetch. For instance, simulate that fetchUsers returns a couple of users (one active, one inactive) and verify the table displays them properly (with inactive one maybe having different appearance and an “Activate” button).
* **Action Handlers Tests:** Simulate clicking the Deactivate button and ensure that fetch is called with correct URL and method. We can use Jest spies on window.fetch. Then simulate the promise resolving and check that the UI updates (user row becomes inactive and Activate button appears). Similar for Activate, Edit, Reset actions. We might not hit the actual network; instead, we intercept the fetch call and return a dummy success response. For example, for reset password, ensure after clicking, a confirmation message appears to the admin.
* **Edit Modal Tests:** Render the UserEditModal with a dummy user and simulate changing a field, clicking Save, and ensure it calls fetch with correct payload. We’ll have to mock the fetch response with updated user JSON and ensure onClose is called with that.
* **Navigation/Route Tests:** Test that unauthorized roles are redirected. For instance, render the UsersPage with currentUser.role = "manager" and expect that it immediately navigates away or shows unauthorized (we might simulate navigate by spying on useNavigate from react-router). Or test that if not logged in, it redirects to login (the UsersPage already does this check in useEffect[[39]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L83-L91)).
* **Audit Trail UI Tests:** Render the UserDetailsPage with a given user ID, mock the fetch for /audit to return a list of events, and ensure that those events are displayed in order. Check that if no events, a message is shown. Also simulate clicking from UsersPage to UserDetails (maybe outside scope of unit test, but a Cypress end-to-end test could cover the link).
* **Customers Page Tests:** Similar approach: test that searching calls the right endpoint and results render. Test editing a customer in the modal triggers the right calls and UI updates. Also test that the “Delete Customer” button calls the GDPR endpoint and perhaps that the customer is then marked deleted in UI (maybe removed from list or shown as “[deleted]” depending on how we implement UI handling of that response).

**End-to-End Testing:** It would be valuable to run a few real-browser end-to-end tests (using Cypress or Playwright) to cover the whole flow: - Log in as an admin in a test environment. - Go to Users page, create a new user (verify it appears in list). - Edit that user’s role and name, then log out and log in as that user to confirm changes took effect (this would test backend as well). - Deactivate the user, attempt login as that user (should fail). - Reactivate, login should succeed again. - Reset password for that user, then try logging in with the new password. - Also test that a manager account cannot see the Users page (e.g., if we have a login for a manager, ensure Users nav is hidden or route blocked).

And similarly for a customer: create a customer via POS or API, then as admin search and edit it, delete it, etc.

These tests ensure that the entire system works as expected and that our multi-tenant and RBAC measures hold up in practice. Any regression (like accidentally allowing a wrong role) would be caught.

### Test Coverage Goals

* **Backend:** Aim for high coverage (at least all critical paths tested). We particularly want to cover the security checks in tests, because those are easy to overlook in development. We’ll also test migration scripts in a staging environment to ensure is\_active default is correctly applied and doesn’t break existing data.
* **Frontend:** We aim to cover the new UI logic with unit tests for each component and integration tests for the flows. While 100% coverage isn’t necessary, the key user flows (create, edit, deactivate, reset, view audit) should be represented in tests.

Additionally, we will perform manual testing of the UI in multiple browsers to ensure modals, tables, etc., render correctly and that the UX is smooth.

By having this robust test suite, we can deploy the new features with confidence. And going forward, any changes to user management will be caught by these tests, ensuring we maintain the integrity of these critical admin functions.

## Conclusion

With the above implementation plan, NovaPOS will significantly enhance its user management system from an MVP-level feature to a more **complete, secure, and user-friendly module**. The backend changes introduce full lifecycle support (create/update/deactivate/reset) with proper audit logging and multi-tenant safeguards, while the frontend changes empower admins with an easy-to-use interface to manage employee accounts and customer profiles. These improvements close the identified gaps[[12]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L73-L79) and align with the product’s requirements for security and operability. Admin users will be able to confidently onboard employees, adjust their roles or access, troubleshoot login issues with password resets, and even maintain customer accounts – all within the Admin Portal, with a clear record of all actions taken.

By implementing these changes, NovaPOS not only addresses immediate needs (like deactivating a departed employee’s access), but also lays groundwork for future scalability: the audit trails and RBAC enforcement will support compliance audits, the architecture will support adding more account types or integrating with an identity provider if needed, and the consistent handling of multi-tenancy will facilitate onboarding many tenants securely. This plan ensures that user management in NovaPOS becomes a robust, enterprise-ready feature, rather than a limiting factor, as the platform grows.

**Sources:**

* NovaPOS MVP Gaps Analysis[[11]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L66-L73)[[12]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L73-L79) (current limitations of user management and audit visibility)
* Implementation Plan for MVP Gaps[[40]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L445-L453)[[25]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L457-L464) (planned enhancements for user management UI and audit logging)
* Auth-Service Code[[1]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/main.rs#L134-L141)[[3]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L393-L401) (existing user API endpoints and tenant scoping)
* Admin Portal Code[[13]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L93-L101)[[6]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L461-L469) (current Users page behavior, listing users without edit options)

[[1]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/main.rs#L134-L141) main.rs

<https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/main.rs>

[[2]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L363-L371) [[3]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L393-L401) [[19]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L29-L37) [[23]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L23-L31) [[24]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L345-L353) [[37]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs#L145-L153) user\_handlers.rs

<https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/auth-service/src/user_handlers.rs>

[[4]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L447-L455) [[16]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L441-L449) [[18]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L457-L461) [[25]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L457-L464) [[27]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L471-L479) [[31]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L439-L446) [[40]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md#L445-L453) Implementation\_Plan\_for\_Addressing\_NovaPOS\_MVP\_Gaps.md

<https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/Implementation_Plan_for_Addressing_NovaPOS_MVP_Gaps.md>

[[5]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/customer-service/src/main.rs#L188-L196) [[20]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/customer-service/src/main.rs#L80-L88) [[21]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/customer-service/src/main.rs#L30-L38) [[22]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/customer-service/src/main.rs#L73-L81) [[38]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/customer-service/src/main.rs#L56-L65) main.rs

<https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/services/customer-service/src/main.rs>

[[6]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L461-L469) [[7]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L262-L271) [[8]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L368-L376) [[9]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L105-L114) [[10]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L116-L124) [[13]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L93-L101) [[14]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L320-L329) [[15]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L76-L83) [[26]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L243-L251) [[28]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L432-L441) [[29]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L444-L452) [[30]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L270-L279) [[32]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L76-L84) [[33]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L146-L154) [[34]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L94-L101) [[35]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L270-L278) [[36]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L277-L284) [[39]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx#L83-L91) UsersPage.tsx

<https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/frontends/admin-portal/src/pages/UsersPage.tsx>

[[11]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L66-L73) [[12]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L73-L79) [[17]](https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md#L76-L78) MVP\_Gaps.md

<https://github.com/datawrangler05/novapos/blob/7f7ec40e7568b98c9c7f4fae84e6c071d7b0230c/doc/analysis/MVP_Gaps.md>