Internal Lab User Management Module Design Specification

This document outlines the design and implementation details for the Internal Lab User Management module. This module enables administrators to manage internal lab staff accounts, including roles, authentication, and access control.

1. Module Purpose

The primary purpose of this module is to provide administrators with the tools to manage all internal lab personnel accounts. This includes creating new user accounts, assigning roles (Admin, Lab Technician, Doctor/Approver, Lab Manager), modifying existing user details, and deactivating/deleting accounts. It also defines the secure authentication mechanism for internal staff and enforces role-based access control (RBAC) across the application.

2. Key Features

- **Role Definition:** Clearly defined roles for internal lab staff:
  - Admin: Full system access, including user management.
  - Lab Technician: Access to sample receiving, lab result entry.
  - Doctor/Approver: Access to lab result approval.
  - Lab Manager: Overview dashboards, reporting, potentially some administrative tasks but not full user management.
- **Secure Authentication (Login/Logout):**
  - Standard email/password login for internal staff.
  - Session management via JWT.
- **User Administration (Admin Role Only):**
  - **Create User:** Add new internal lab staff accounts with initial role assignment.
  - **View Users:** List all internal lab users with their details and roles.
  - **Edit User:** Modify user details (e.g., email, name, role).
  - **Delete User:** Remove user accounts.
- **Authorization (Role-Based Access Control - RBAC):**
  - Implement granular access control based on user roles, restricting access to specific pages, features, and API endpoints.

3. Frontend Design Specifications (React with React Router & Tailwind CSS)

A. Pages & Components

1. **/auth/login (Login Page):**
   - **Layout:** Shared with customer login, typically an AuthLayout.

- **Components:** LoginForm (as described in customer-registration-module-spec).
- **State Management:** React Hook Form, Zod, React Query (useMutation).
- **Interactions:**
  - On successful login, based on the role returned in the JWT, redirect to the appropriate dashboard (e.g., /admin/dashboard, /lab/dashboard, /doctor/approval, /customer/dashboard).
  - Display clear error messages for invalid credentials or unconfirmed email.

2. ***/admin/users (User Management - List Page):***
   - **Layout:** AdminLayout with sidebar navigation.
   - **Components:**
     - UserTable component:
       - Displays a paginated list of internal users (ID, Email, Role, Created At, Actions).
       - Search/filter bar (Shadcn UI Input).
       - "Add User" button (Shadcn UI Button).
     - Table component (Shadcn UI Table): For displaying user data.
     - Pagination component (Shadcn UI Pagination).
   - **State Management:**
     - React Query (useQuery): Fetch list of users. Manage pagination and filtering state within the query key.
     - Local state for search query, current page, etc.
   - **Interactions:**
     - Clicking "Add User" opens UserForm in a modal or navigates to a new page.
     - "Edit" button (per row) opens UserForm with pre-filled data.
     - "Delete" button (per row) triggers a confirmation dialog (custom AlertDialog using Shadcn UI AlertDialog).
     - Show loading states for data fetching and mutations.
     - Display success/error toasts.
3. ***/admin/users/add or UserForm (Modal/Page for Add/Edit User):***
   - **Layout:** Can be a standalone page or a modal within the /admin/users page.
   - **Components:**
     - UserForm component:
       - Input fields (Shadcn UI Input): Email, Password (only for add), Confirm Password (only for add).
       - Select/Dropdown (Shadcn UI Select): Role (options: Admin, Lab Technician, Doctor, Lab Manager).
       - Submit/Save button (Shadcn UI Button).

    - **State Management:**
        - React Hook Form: Manage form state and validation.
        - Zod: Define schema for client-side validation.
        - React Query (useMutation): For adding (POST) or updating (PUT) user data.
    - **Interactions:**
        - For "Add": Clear fields, require email and password.
        - For "Edit": Pre-fill fields with existing user data (email may not be editable, or require separate email change flow), password fields optional (if not changing password).
        - Show loading state during submission.
        - Display validation errors.
        - Close modal/redirect to user list on success.

B. Routing (React Router)

- path: '/admin/users' (Protected route, requires admin role)
    - loader to fetch user list.
- path: '/admin/users/add' (Optional, if "Add User" is a separate page)
    - action to handle user creation.
- path: '/admin/users/:userId' (Optional, for detailed user view or editing as a separate page)
    - loader to fetch single user details.
    - action to handle user updates/deletion.

C. Styling

- **Tailwind CSS:** Consistent application of utility classes for layout, spacing, typography, and responsive design.
- **Shadcn UI:** Used for all UI components (tables, forms, buttons, modals, alerts). Ensure components are styled consistently.
- **Responsive Design:** Tables should be scrollable horizontally on smaller screens if necessary, or collapse into card-like views. Forms should adjust elegantly.

D. Validation

- **Client-side (Zod + React Hook Form):**
    - Email format validation.
    - Password strength (min length, complexity requirements).
    - Role selection is required.
    - Password and confirm password match (for create/change password).
- **Server-side (Zod in Express):** Re-validate all incoming data for security and data

integrity. Crucial for ensuring that only valid roles are assigned.

4. Backend Design Specifications (Express.js with TypeScript, PostgreSQL, Prisma)

A. Relevant Database Tables (from lab-tracking-webapp-plan immersive)

- users table: Stores id, email, password_hash, role, is_email_confirmed, created_at, updated_at.
- audit_trail table: For logging all user management actions (create, update, delete).

B. API Endpoints

1. **POST /api/v1/auth/login (Internal Staff Login)**
   - **Purpose:** Authenticate internal staff and issue JWT.
   - **Request/Response:** Same as customer login.
   - **Logic:** After successful email/password validation, generate JWT with userId and role. The frontend will redirect based on this role.
2. **GET /api/v1/admin/users (List Internal Users)**
   - **Purpose:** Retrieve a list of all internal lab users.
   - **Request:** Optional query parameters for pagination (page, limit), search (searchQuery), and filtering (role).
   - **Response (200 OK):** { data: User[], total: number, page: number, limit: number } (User objects should exclude password_hash).
   - **Errors (403 Forbidden):** If the authenticated user is not an admin.
   - **Middleware:** authMiddleware, roleMiddleware('admin'), Input validation (for query params).
   - **Logic:**
     - Extract userId from JWT, verify admin role.
     - Query users table, applying pagination, search, and role filters.
     - Exclude password_hash from results.
     - Log action to audit_trail (e.g., 'admin_viewed_users').
3. **GET /api/v1/admin/users/:id (Get Single User Details)**
   - **Purpose:** Retrieve details of a specific internal lab user.
   - **Request:** URL parameter :id (UUID of the user).
   - **Response (200 OK):** User object (excluding password_hash).
   - **Errors (404 Not Found):** If user does not exist.
   - **Middleware:** authMiddleware, roleMiddleware('admin').
   - **Logic:**
     - Extract userId from JWT, verify admin role.
     - Fetch User record by id.

        - Exclude password_hash.
        - Log action to audit_trail (e.g., 'admin_viewed_user_details').
4. **POST /api/v1/admin/users (Create New Internal User)**
    - **Purpose:** Create a new account for internal lab staff.
    - **Request:** application/json
        - Body: email, password, role (must be one of the predefined internal roles: lab_technician, doctor, lab_manager, admin).
    - **Response (201 Created):** { message: 'User created successfully.', userId: '...' }
    - **Errors (400 Bad Request):** If email already exists, invalid role, validation errors.
    - **Middleware:** authMiddleware, roleMiddleware('admin'), Input validation (Zod schema for email, password, role).
    - **Logic:**
        - Extract userId from JWT, verify admin role.
        - Validate input, especially ensuring role is a valid internal role.
        - Hash password (bcrypt).
        - Create User entry with is_email_confirmed: true (internal users don't need email confirmation for initial login, can be handled by admin).
        - Log action to audit_trail (e.g., 'admin_created_user', details: new user's email, role).
5. **PUT /api/v1/admin/users/:id (Update Internal User Details)**
    - **Purpose:** Update an existing internal user's details, including their role.
    - **Request:** application/json
        - Body: email (optional), role (optional), password (optional, for changing password).
    - **Response (200 OK):** { message: 'User updated successfully.' }
    - **Errors (400 Bad Request):** Validation errors, trying to change own role (if not permitted), attempting to delete last admin.
    - **Middleware:** authMiddleware, roleMiddleware('admin'), Input validation.
    - **Logic:**
        - Extract userId from JWT, verify admin role.
        - Fetch existing user by :id.
        - Validate incoming email (if changed, must be unique), role (must be valid internal role).
        - If password is provided, hash it.
        - Update User record.
        - Consider adding a check: an admin should not be able to demote/delete the *last* admin account.
        - Log action to audit_trail (e.g., 'admin_updated_user', details: user ID,

old/new roles, changed fields).

6. **DELETE /api/v1/admin/users/:id (Delete Internal User)**
   - **Purpose:** Delete an internal user account.
   - **Request:** None (uses URL parameter :id).
   - **Response (204 No Content):** (Successful deletion)
   - **Errors (400 Bad Request):** Trying to delete own account, trying to delete the last admin account.
   - **Middleware:** authMiddleware, roleMiddleware('admin').
   - **Logic:**
     - Extract userId from JWT, verify admin role.
     - Prevent admin from deleting their *own* account or the *last* active admin account.
     - Delete User record. Prisma's cascading deletes (if configured) will handle related records in customers (if a customer user), or ensure no direct dependencies from lab staff exist.
     - Log action to audit_trail (e.g., 'admin_deleted_user', details: deleted user's email, role).

C. Prisma Operations

- PrismaClient.user.findMany() (for list, with pagination/filtering)
- PrismaClient.user.findUnique() (for single user details, pre-fill edit form)
- PrismaClient.user.create() (for adding new user)
- PrismaClient.user.update() (for editing user details)
- PrismaClient.user.delete() (for deleting user)
- PrismaClient.auditTrail.create()

D. Middleware Requirements

- authMiddleware: Verifies JWT from Authorization header and populates req.user with { id, role }.
- roleMiddleware('admin'): Ensures the authenticated user's role is 'admin' for all administrative actions. Can be extended to roleMiddleware(['admin', 'lab_manager']) for less sensitive tasks.
- validateMiddleware(schema): For validating request bodies for POST and PUT operations, and query parameters for GET operations.

E. Business Logic / Service Functions

- UserService:
  - createUser(email, password, role): Handles password hashing and user creation.
  - getUsers(filters): Handles querying and filtering users.

    - getUserById(id): Retrieves a single user.
    - updateUser(id, data): Handles updating user details.
    - deleteUser(id): Handles user deletion.
    - checkAdminCount(): Utility to ensure at least one admin account exists before deletion/demotion.
- AuthService: Manages JWT generation and verification (used by authMiddleware).
- AuditService: For logging all user management actions.

F. Error Handling

- Custom error classes (e.g., NotFoundError, ForbiddenError, ConflictError for duplicate email, ValidationError).
- Centralized Express error handling middleware to catch these errors and return appropriate HTTP status codes (e.g., 400, 401, 403, 404, 500).

5. Shared Types (from packages/shared/types.ts)

Define interfaces for:

- User (id, email, role, isEmailConfirmed, createdAt, updatedAt).
    - Role Enum/Union type: 'customer' | 'admin' | 'lab_technician' | 'doctor' | 'lab_manager'.
- CreateUserPayload: { email: string, password: string, role: Role }.
- UpdateUserPayload: { email?: string, password?: string, role?: Role }.
- UserFilters: { searchQuery?: string, role?: Role, page?: number, limit?: number }.
- PaginatedResponse<T>: { data: T[], total: number, page: number, limit: number }.

6. Edge Cases & Considerations

- **Security:**
    - Ensure robust password hashing.
    - JWTs should have reasonable expiration times and be refreshed securely.
    - Strict RBAC enforcement on the backend is paramount. Never trust client-side role checks.
    - Prevent admins from inadvertently locking themselves out (e.g., by deleting the last admin account).
- **Audit Trail:** Every significant action (create, update, delete user, role changes) must be logged in the audit_trail table.
- **User Experience:**
    - Clear feedback on user actions (e.g., "User added successfully," "Error deleting user").
    - Confirmation dialogs for destructive actions (e.g., deleting a user).

    - Form validation messages that guide the user.
- **Password Reset for Internal Users:** While not explicitly in "User Administration," a "Forgot Password" flow will likely be needed for internal users too. This might involve sending a temporary link or code to their email.
- **Pagination & Filtering:** Implement server-side pagination and filtering for user lists to ensure performance with large datasets.
- **Default Roles:** Decide on a default role for newly created internal users if not explicitly specified by the admin.