Fintech Payment Management System – Code Logic & Walkthrough

1. Overview

The Fintech Payment Management Dashboard is a Vue 3 Single Page Application (SPA) designed for internal financial operations and recording payments and users who are using this alongwith their roles and date they joined the platform . It features a responsive user interface for managing users and payments with role-based access control (RBAC). Core functionality is implemented using Pinia stores, mock local storage data, and Vue Router for navigation.

Key roles: Admin, Manager, Staff.

2. Technical Architecture & Stack

- **Frontend Framework**: Vue 3 with Composition API (`<script setup>`).
- **State Management**: Pinia stores (`UserStore.js`, `PaymentStore.js`) for centralized reactive state.
- **Routing**: Vue Router (`index.js`) handles SPA navigation. Supports route parameters for editing/viewing specific records.
- **Persistent Storage**: LocalStorage is used to mock backend persistence.
- **Styling**: Scoped CSS per component, dark-themed UI, responsive tables and forms.
- **Testing**: Unit tests using **Vitest** and **Vue Test Utils** .

3. Routing Logic (`router/index.js`)

- **Root redirect**: `/` → `/dashboard`.
- **Dashboard**: `/dashboard` lazy-loaded.
- **User Routes**:
  - `/users` → `UserList.vue`
  - `/users/new` → `UserForm.vue` (Add)
  - `/users/edit/:id` → `UserForm.vue` (Edit and delete with props)
- **Payment Routes**:
  - `/payments` → `PaymentList.vue`
  - `/payments/new` → `PaymentForm.vue`
  - `/payments/edit/:id` → `PaymentForm.vue` (Edit)
  - `/payments/:id` → `PaymentDetail.vue` (View single payment)

Navigation is reactive, and route parameters determine **edit vs add mode** for forms.

4. User Management

4.1 User Store (`UserStore.js`)

- **State**: `users` array, `nextUserId` auto-increment.

- **Persistent Storage**: `localStorage` used for saving users and next ID.
- **Actions**:
  - `addUser(user)`: Adds new user with auto-generated ID, default role `'Staff'`.
  - `updateUser(user)`: Updates user by ID, sets `updatedAt`.
  - `deleteUser(id)`: Deletes a user by ID.
  - `persist()`: Saves state to `localStorage`.

Initialization: Loads users from `localStorage` .

4.2 User Form (`UserForm.vue`)

- Determines mode based on route param (`isEdit`).
- **Add Mode**: Sets `createdAt` to current date, submits new user to store.
- **Edit Mode**: Populates form from existing user; allows updating and deletion.
- Form uses **v-model** for two-way binding.
- Buttons:
  - Submit → Add or Update
  - Delete → Removes user after confirmation

4.3 User List (`UserList.vue`)

- Displays table of users with filtering:
    - `Created From` (date)
    - `Role` (select unique roles)
- Computed property `filteredUsers` applies filters dynamically.
- Role badges:
    - Admin → Blue
    - Manager → Orange
    - Staff → Green
- Edit button: Red-styled, links to `/users/edit/:id`.
- Add button: Blue-styled, links to `/users/new`.

5. Payment Management

The Payments module allows users to **view, add, edit, and filter payments**. It interacts with the **PaymentStore** and references users via **UserStore** for displaying user names.

5.1 Payment Store (`PaymentStore.js`)

- **State**:
    - `payments`: Array of all payments.
    - `nextPaymentId`: Auto-incremented ID for new payments.
- **Initialization**: Loads from `localStorage` or defaults with sample payments:

Actions:

- `addPayment(payload)`: Adds a payment, resolves `user` name from `userId` if necessary. Updates localStorage.
- `updatePayment(payload)`: Updates payment fields. Ensures `user` name stays consistent with `userId`.
- `getPaymentById(id)`: Returns a single payment by ID.
- `persist()`: Saves `payments` array and `nextPaymentId` to `localStorage`.

5.2 Payment List (`PaymentList.vue`)

Functionality:

- Displays **all payments** in a table with **filters**:
    - `Status`: Pending / Completed / Failed

    - `Type`: Salary / Subscriptions / Vendor Payments / Client Payments
    - `Min Amount`: Filter payments greater than or equal to input
    - `Date From`: Filter payments after a specific date
- **Reactive filtering** is handled via **computed property**:

- **User display**: Shows `payment.user` if available; otherwise resolves name via `getUserName(userId)` from `UserStore`.
- **Actions per row**:
    - `View` → Navigates to `/payments/:id` (`PaymentDetail.vue`).
    - `Edit` → Navigates to `/payments/edit/:id` (`PaymentForm.vue`).
- **Add new payment**: Button navigates to `/payments/new`.
- **Status styling**: Conditional CSS classes for **Pending / Completed / Failed** with color-coded badges for clarity.

5.3 Payment Detail (`PaymentDetail.vue`)

Functionality:

- Displays **detailed payment information**:
    - ID, User, Amount, Type, Status, Payment Date.
- Resolves `user` name dynamically if missing (`getUserName(payment.userId)`).
- Provides **Back to Payments** button.

5.4 Payment Form

While `PaymentForm.vue` was not provided, based on `UserForm.vue` logic, we can assume:

- **Add Mode**:
    - Sets default `status` to `'Pending'`.
    - Resolves `user` from `userId`.
    - Auto-assigns `id` using `nextPaymentId`.
- **Edit Mode**:
    - Loads payment by `id` from `PaymentStore`.
    - Allows editing of `amount`, `type`, `status`, and `userId`.
- On submit, calls `addPayment` or `updatePayment`.

Buttons:

- Submit → Add / Update payment.
- Optional Delete → Removes payment after confirmation.

5.5 Filters and Computed Properties

- Filters are **reactive** via `ref` values for `status`, `type`, `amount`, and `date`.
- Computed `filteredPayments` updates **in real-time** without additional API calls.
- This ensures **instant feedback** as users adjust filters.

5.6 UI & UX

- **Responsive table** with `display: block` on mobile.
- **Hover effects** for table rows.
- **Colored badges** for payment status.
- **Action buttons** clearly distinguish between View and Edit actions.
- **Add New Payment** button fixed at bottom for easy access.

6. Dashboard & Summary Cards

The **Dashboard** provides an overview of the system's key metrics at a glance, including the total number of users, total payments, and a breakdown of payment statuses.

Key Features

1. **Summary Cards**
    - **Total Users:** Displays the number of registered users.
    - **Total Payments:** Shows the total number of payments recorded.
    - **Completed / Pending / Failed:** Cards with visual distinction for payment statuses. Each card uses color coding to quickly identify metrics:
        - Green for completed
        - Orange for pending
        - Red for failed

2. **Action Buttons**
   - **View Users:** Navigates to the users management page.
   - **View Payments:** Navigates to the payments management page.
3. **Responsive Design**
   - The dashboard adjusts gracefully to mobile and tablet screens. Summary cards rearrange to fit smaller widths, ensuring readability.

The dashboard is linked to **Pinia stores** (`UserStore` and `PaymentStore`) to dynamically compute values using `computed` properties. This ensures that any changes in users or payments are immediately reflected in the dashboard.

7. Testing

Testing ensures the correctness of both **stores** and **components**.

7.1 Store Tests

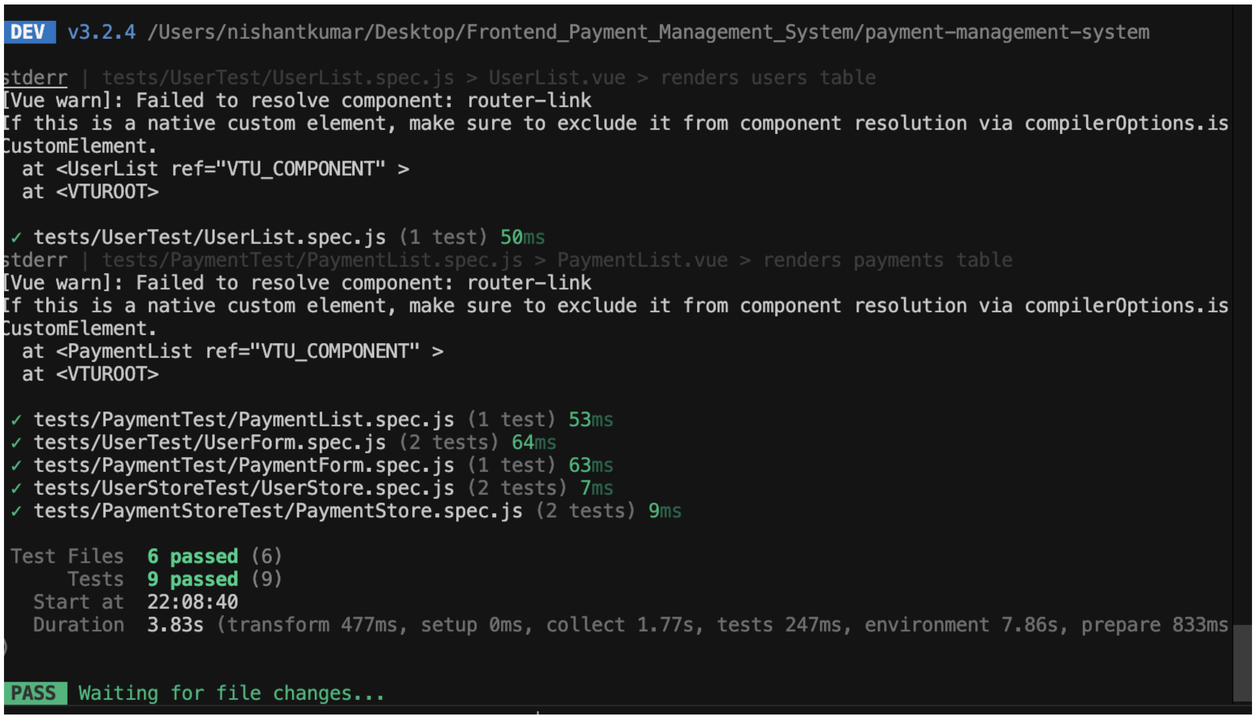
- **PaymentStore:**
  - **Add Payment:** Verifies that a new payment is added and data is updated correctly.
  - **Update Payment:** Ensures updating a payment's status modifies the store correctly.
- **UserStore:**
  - **Add User:** Confirms new users are added to the store.

    - **Update User:** Checks that user modifications (e.g., role) persist in the store.

7.2 Component Tests

- **PaymentList.vue & UserList.vue:**
    - Verifies that tables render with correct headings and rows.
- **PaymentForm.vue & UserForm.vue:**
    - Checks that forms render correctly in **add** and **edit** modes.
    - Ensures input fields and buttons are present.
- **PaymentDetail.vue:**
    - Confirms detailed information for each payment is displayed correctly.

These tests use **Vitest**, **Vue Test Utils**, and **Pinia Testing** to simulate state and user interactions.



```
DEV  v3.2.4 /Users/nishantkumar/Desktop/Frontend_Payment_Management_System/payment-management-system

stderr | tests/UserTest/UserList.spec.js > UserList.vue > renders users table
[Vue warn]: Failed to resolve component: router-link
If this is a native custom element, make sure to exclude it from component resolution via compilerOptions.is
CustomElement.
  at <UserList ref="VTU_COMPONENT" >
  at <VTUROOT>

 ✓ tests/UserTest/UserList.spec.js (1 test) 50ms
stderr | tests/PaymentTest/PaymentList.spec.js > PaymentList.vue > renders payments table
[Vue warn]: Failed to resolve component: router-link
If this is a native custom element, make sure to exclude it from component resolution via compilerOptions.is
CustomElement.
  at <PaymentList ref="VTU_COMPONENT" >
  at <VTUROOT>

 ✓ tests/PaymentTest/PaymentList.spec.js (1 test) 53ms
 ✓ tests/UserTest/UserForm.spec.js (2 tests) 64ms
 ✓ tests/PaymentTest/PaymentForm.spec.js (1 test) 63ms
 ✓ tests/UserStoreTest/UserStore.spec.js (2 tests) 7ms
 ✓ tests/PaymentStoreTest/PaymentStore.spec.js (2 tests) 9ms

 Test Files  6 passed (6)
      Tests  9 passed (9)
   Start at  22:08:40
   Duration  3.83s (transform 477ms, setup 0ms, collect 1.77s, tests 247ms, environment 7.86s, prepare 833ms
)

 PASS  Waiting for file changes...
```

8. Setup & Repository

8.1 Project Setup

Clone the repository:

```
git clone <repo-link>
cd <project-folder>
```

Install dependencies: `npm install`
Run the development server: `npm run dev`
Run tests: `npm run test`

8.2 Repository ::: [Github Repo link]