# Design, Implementation and Testing

## for

# Web3 Campus Marketplace

**Prepared by**

Yang Yanqi
Fan QianYi
Qiu Yixuan
Huang Qiyuan
Hu ShengQuan
Zhang Song

**G7**

**2026.01.24**

# Table of Contents

# 1. System Architecture

## 1.1 Architectural Overview

The Web3 Campus Marketplace is engineered upon a Modular Three-Tier Architecture, integrated with a specialized Web3 Simulation Service Layer. This design choice bridges standard web application paradigms with decentralized identity (DID) concepts, creating a hybrid environment tailored for the university ecosystem.

The system serves as a secure, verified trading platform where traditional authentication mechanisms coalesce with simulated blockchain interactions. By decoupling the presentation logic from the business rules and data persistence, the architecture ensures scalability and maintainability. A distinct feature of this architecture is the Simulated Trust Model, which mimics on-chain cryptographic operations—specifically wallet signing and transaction verification—within a controlled off-chain environment. This approach allows for the practical demonstration of Web3 principles without the operational overhead or financial risks associated with live mainnet connectivity. As Figure 1 shown

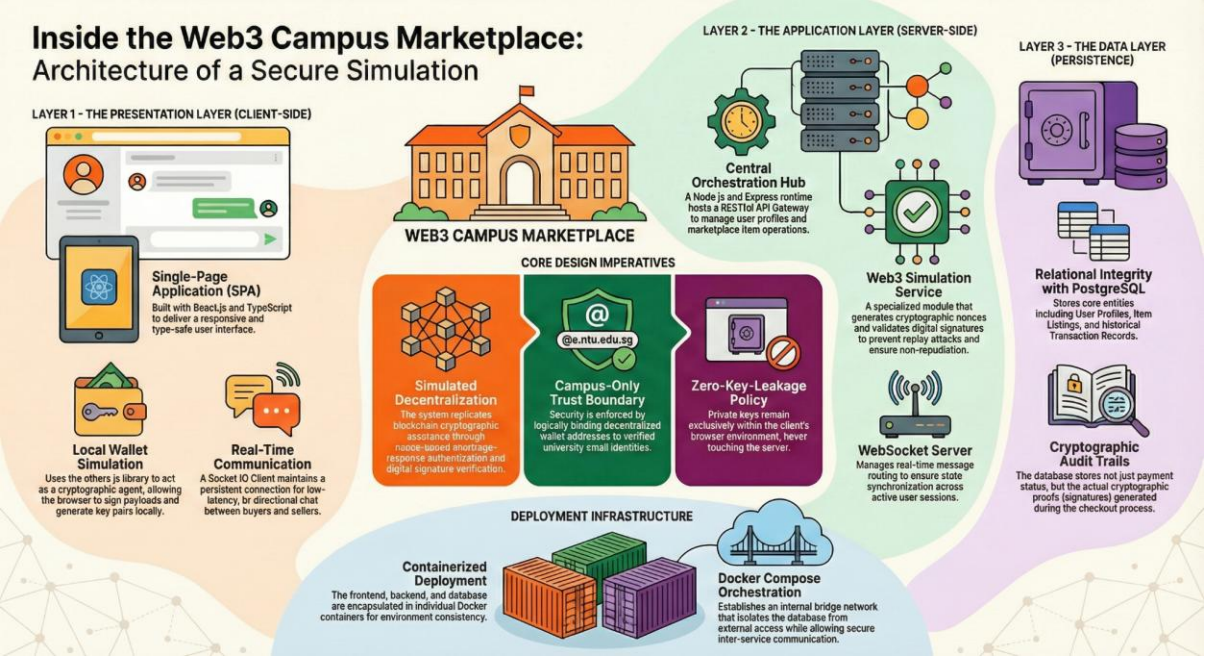


Fig.1.1 Web3 Marketplace Architectural Overview

## 1.2 Design Imperatives and Constraints

The architectural decision-making process is driven by specific functional requirements and constraints outlined in the Software Requirements Specification (SRS).

The primary design imperative is Simulated Decentralization, which mandates that the system must authentically replicate the cryptographic assurance of blockchain transactions. This requires the architecture to support nonce-based challenge-response authentication and digital signature verification, treating the user's local wallet as the root of trust rather than server-side sessions alone. Concurrently, the system enforces a Campus-Only Trust Boundary, logically binding decentralized wallet addresses to verified university email identities (e.g., @e.ntu.edu.sg) to mitigate fraud.

Technologically, the implementation adheres to a strict stack: React.js (TypeScript) for the client-side interface, Node.js/Express for server-side logic, and PostgreSQL for relational persistence. Operational constraints dictate a "Zero-Key-Leakage" policy, ensuring that private keys remain exclusively within the client's browser environment. Furthermore, to facilitate seamless integration and testing, the entire system infrastructure is containerized, ensuring consistent deployment across diverse development environments.

## 1.3 Logical Architecture Decomposition

The system is partitioned into three logical layers, each with distinct responsibilities and interfaces.

### 1.3.1 Presentation Layer (Client-Side)

The client interface is delivered as a Single-Page Application (SPA) utilizing React.js. This layer is responsible not only for rendering the user interface but also for managing the Local Wallet Simulation. By integrating the ethers.js library, the client acts as a cryptographic agent, generating key pairs and signing payloads (such as login challenges and transaction intents) locally. Additionally, a Socket.IO Client maintains a persistent, asynchronous connection to the backend, enabling low-latency, bi-directional communication for the chat module.

### 1.3.2 Application Layer (Server-Side)

The Application Layer, hosted on a Node.js/Express runtime, functions as the central orchestration hub. It exposes a RESTful API Gateway to handle synchronous HTTP requests for user management and item CRUD operations. Embedded within this layer is the Web3 Simulation Service, a specialized module that encapsulates the logic for generating cryptographic nonces and validating digital signatures, thereby preventing replay attacks and ensuring non-repudiation. Parallel to the HTTP services, a WebSocket server manages real-time message routing between buyers and sellers, ensuring state synchronization across active sessions.

### 1.3.3 Data Layer (Persistence Strategy)

Data persistence is managed by PostgreSQL, utilizing a relational schema to maintain data integrity. The database stores critical entities including User Profiles (linking wallet addresses to verification status), Item Listings, and Transaction Records. Notably, the transaction ledger records not just the payment status but also the cryptographic proofs (signatures) generated during the simulated checkout process, supporting the system's auditability requirements.

### 1.4 Deployment Infrastructure

To ensure environment consistency and simplify the demonstration workflow, the system adopts a Containerized Deployment Strategy. The application components—frontend static assets, backend application server, and the database engine—are encapsulated in individual Docker containers. These containers are orchestrated via Docker Compose, establishing an internal bridge network that isolates the database from external access while allowing inter-service communication. This configuration ensures that the simulation environment remains reproducible on any host machine, from local development laptops to university lab servers.

### 1.5 System Workflows and Traceability

The proposed architecture provides the structural foundation for the detailed interaction designs presented in the subsequent Sequence Diagrams.

The Secure Authentication Flow is supported by the interaction between the client-side wallet and the server-side Web3 Service, implementing a challenge-response mechanism. Similarly, the Transaction Signing Flow leverages the decoupled nature of the architecture: the backend constructs transaction payloads, while the frontend handles the signing process. This separation ensures that the system accurately simulates the "trustless" nature of blockchain interactions

## 2. Application Skeleton

```
web3-campus-marketplace/
├── docker-compose.yml        # Orchestration for Frontend, Backend, DB
├── .env                      # Environment variables (DB_URL, JWT_SECRET, etc.)
├── client/                   # LAYER 1: PRESENTATION LAYER (React.js)
│   ├── Dockerfile
│   ├── src/
│   │   ├── assets/
│   │   ├── components/       # Reusable UI components
│   │   ├── services/         # API adapters & Socket.IO client
│   │   ├── wallet/           # Local Wallet Simulation (ethers.js wrapper)
│   │   ├── pages/            # Marketplace, Profile, ListingDetail
│   │   ├── types/            # TypeScript Interfaces (User, Listing, Tx)
│   │   └── App.tsx
│   └── package.json
├── server/                   # LAYER 2: APPLICATION LAYER (Node.js/Express)
│   ├── Dockerfile
│   ├── src/
│   │   ├── config/           # DB config, Constants
│   │   ├── controllers/      # Request handlers
│   │   ├── middleware/       # Auth, Validation
```

```
│   │   ├──── models/          # ORM/DAO models
│   │   ├──── routes/          # RESTful API definition
│   │   ├──── services/
│   │   │   ├──── Web3Service.ts  # Crypto nonces & signature validation
│   │   │   └──── WebSocket.ts    # Real-time message routing
│   │   └──── app.ts           # Entry point
│   └──── package.json
└──── db/                  # LAYER 3: DATA LAYER (PostgreSQL)
    ├──── Dockerfile
    └──── init.sql           # Database Schema Initialization
```

# 3. Class Diagram

## 3.1 Overview

The Class Diagram illustrates the static structure of the Web3 Campus Marketplace system by modeling the core domain entities, their attributes, operations, and relationships. This diagram serves as the blueprint for the system's object-oriented design, ensuring consistency between the data model, business logic, and the underlying relational database schema. It directly supports the functional requirements defined in the SRS and the architectural patterns described in the System Architecture section.

The design follows a Domain-Driven Design (DDD) approach, focusing on the key aggregates within the campus trading context. The primary entities include User, ItemListing, Transaction, Message, and ForumPost. Special attention is given to modeling the simulated Web3 constructs, such as WalletIdentity and DigitalSignature, which encapsulate the decentralized authentication and transaction logic without a live blockchain dependency. The relationships enforce critical business rules, such as a User must be verified to create Listings, and a Transaction must involve a Buyer and a Seller.

## 3.2 Class Descriptions

### 3.2.1 Core Domain Entities

**1. User**

**Description**: Represents a registered member of the platform. This is the central actor encompassing buyers, sellers, and administrators. The class encapsulates both traditional account information and Web3 identity attributes.

**Attributes**:

userId: UUID (Primary Key)

walletAddress: String (Unique, Not Null) – The public address from the simulated wallet.

campusEmail: String (Unique, Not Null) – Must have a verified university domain (e.g., @e.ntu.edu.sg).

isEmailVerified: Boolean – Flag indicating completion of campus verification.

role: Enum(UserRole) – Values: STUDENT, ADMIN.
username: String
profileImageUrl: String
createdAt: Timestamp

**Operations:**

+authenticate(challenge: String): Signature – Initiates the signing process via the associated WalletIdentity.
+updateProfile(profileData: ProfileDTO): Boolean
+verifyCampusEmail(token: String): Boolean

**Relationships:**

1 User has 1 WalletIdentity (Composition).
1 User creates 0..* ItemListing.
1 User (as Seller) involved in 0..* Transaction.
1 User (as Buyer) involved in 0..* Transaction.
1 User sends/receives 0..* Message.

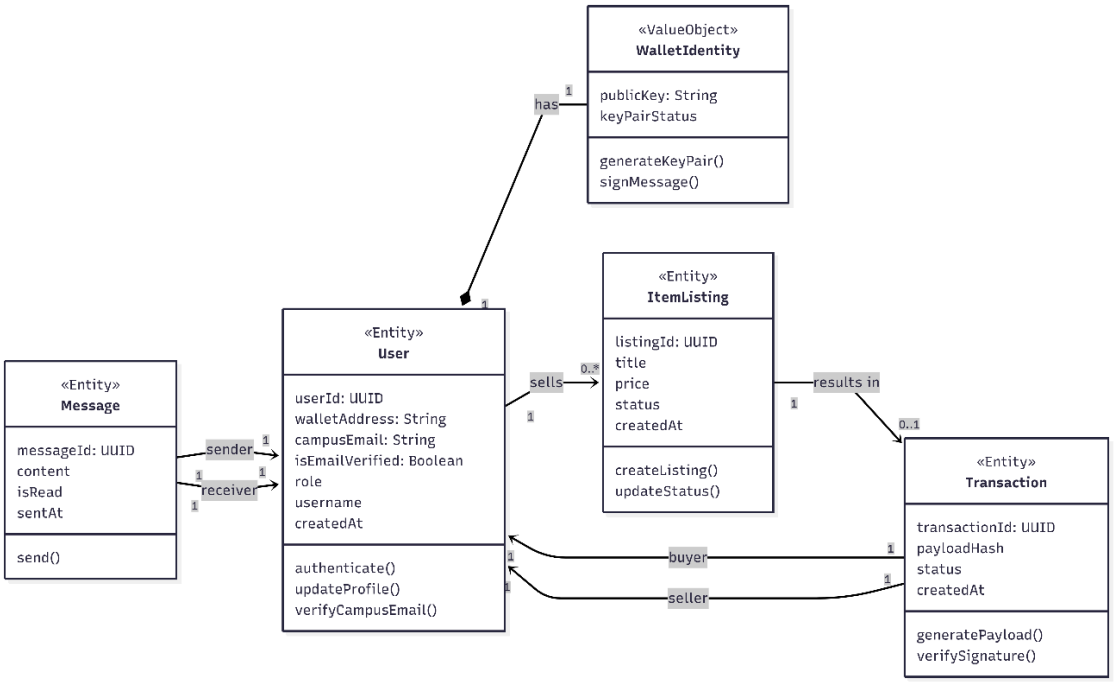


Fig. 3.1 Core Domain Class Diagram (Simplified View)

**2. WalletIdentity**

**Description**: A value object that manages the simulated Web3 cryptographic identity for a User. It is responsible for key pair generation and signing operations, adhering to the "Zero-Key-Leakage" policy by performing all sensitive operations client-side.

**Attributes**:

publicKey: String (Derived from User.walletAddress).
keyPairStatus: Enum(KeyStatus) – e.g., GENERATED, COMPROMISED.

**Operations:**

+generateKeyPair(): void – Called by the frontend upon first login.
+signMessage(message: String): DigitalSignature – Uses the locally stored private key (simulated) to create a signature.

**Relationships:**
1 WalletIdentity belongs to 1 User (Composition).

**3. ItemListing**

**Description**: Represents an item offered for sale on the marketplace. It is the central commodity entity and enforces rules about ownership and state transitions.
**Attributes**:
listingId: UUID (Primary Key)
title: String (Not Null)
description: Text
price: Decimal (Not Null)
category: Enum(ItemCategory) – e.g., BOOKS, ELECTRONICS, FURNITURE.
status: Enum(ListingStatus) – AVAILABLE, PENDING, SOLD, DELETED.
imageUrls: Array<String>
createdAt: Timestamp
updatedAt: Timestamp
**Operations:**
+createListing(seller: User, details: ListingDTO): ItemListing
+updateStatus(newStatus: ListingStatus): Boolean
+isOwnedBy(user: User): Boolean
**Relationships:**
1 ItemListing is owned by 1 User (as Seller).
1 ItemListing is subject of 0..1 Transaction (when sold).

**4. Transaction**

**Description**: Models a simulated purchase agreement between a buyer and a seller. It records the intent, the cryptographic proof of agreement (signature), and the lifecycle state, serving as an audit log for the trading process.
**Attributes**:
transactionId: UUID (Primary Key)
payloadHash: String – Hash of the transaction details (itemId, price, parties) generated by the backend.
buyerSignature: String – The digital signature from the buyer's wallet on the payload hash.
status: Enum(TransactionStatus) – CREATED, SIGNED, CONFIRMED, DISPUTED, COMPLETED.
createdAt: Timestamp
confirmedAt: Timestamp
**Operations:**
+generatePayload(): String – Constructs the structured data string to be signed.
+verifySignature(signature: String, walletAddress: String): Boolean – Validates the signature against the stored payload hash and buyer's address.
**Relationships:**
1 Transaction involves 1 User (as Buyer).
1 Transaction involves 1 User (as Seller).
1 Transaction is for 1 ItemListing.
1 Transaction has 1 DigitalSignature (Aggregate of the buyerSignature).

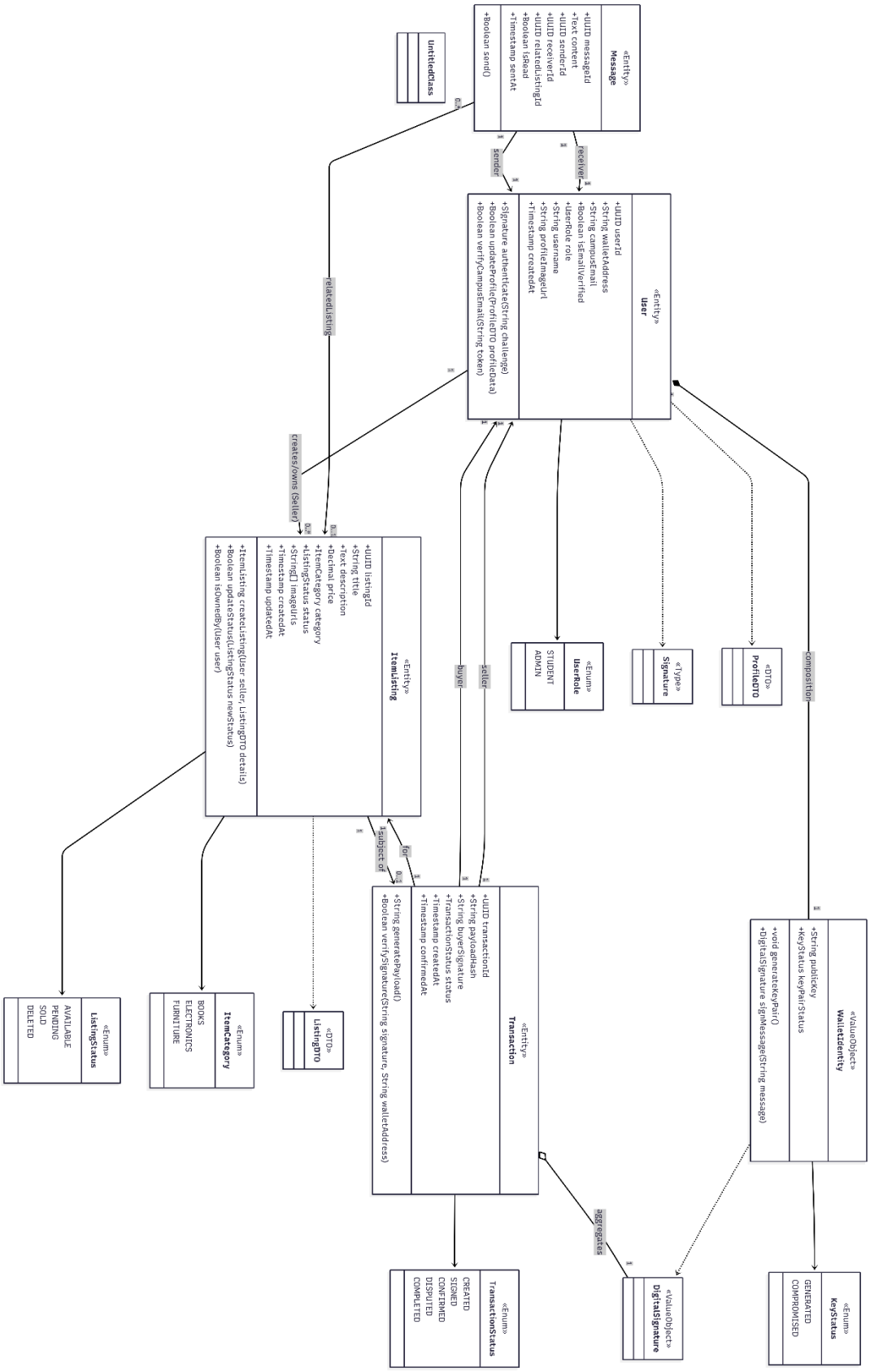


Fig. 3.2 Core Domain Class Diagram (Detailed View)

**5. Message**

**Description**: Represents a single chat message within a real-time conversation between a buyer and a seller related to a specific item or transaction.

**Attributes:**

messageId: UUID (Primary Key)
content: Text (Not Null)
senderId: UUID (Foreign Key to User)
receiverId: UUID (Foreign Key to User)
relatedListingId: UUID (Foreign Key to ItemListing, Optional)
isRead: Boolean
sentAt: Timestamp

**Operations:**

+send(): Boolean – Persists the message and triggers real-time broadcast via WebSocket.

**Relationships:**

1 Message is sent by 1 User.
1 Message is received by 1 User.

### 3.2.2 Supporting & Web3 Simulation Entities

**1. DigitalSignature**

**Description**: A value object that encapsulates the result of a cryptographic signing operation. It is used to validate the authenticity and integrity of both login challenges and transaction payloads.

**Attributes**:

signature: String – The raw signature string.
signedData: String – The original data (challenge or transaction hash) that was signed.
signerAddress: String – The wallet address of the signer.
timestamp: Timestamp

**Operations:**

+validate(): Boolean – Uses the ethers.js library (simulated) to verify the signature matches the data and signer.

**2. ForumPost**

**Description**: Represents a thread in the community forum, enabling social interaction beyond direct transactions.

**Attributes**:

postId: UUID
title: String
content: Text
category: Enum(ForumCategory)
authorId: UUID (Foreign Key to User)
createdAt: Timestamp
Relationships:
1 ForumPost is authored by 1 User.
1 ForumPost has 0..* ForumComment.

**3. ForumComment**

**Description**: A reply to a ForumPost.
**Attributes**:
commentId: UUID
content: Text
authorId: UUID
postId: UUID
createdAt: Timestamp
Relationships:
1 ForumComment belongs to 1 ForumPost.
1 ForumComment is authored by 1 User.

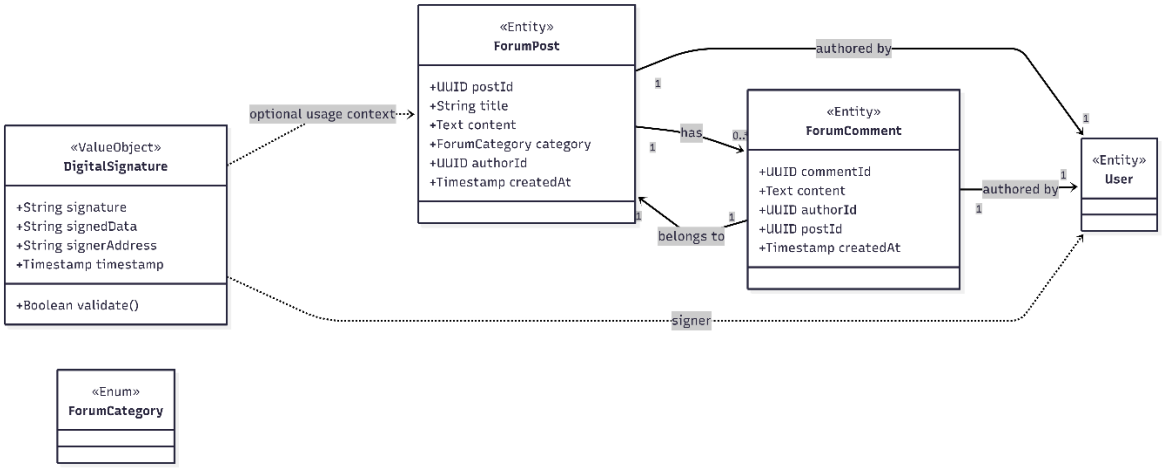


Fig. 3.2 Supporting and Web3 Simulation Entities Class Diagram

## 3.3 Relationships and Multiplicity

The relationships are defined with precise multiplicity to reflect business constraints:

1. **User ↔ WalletIdentity: 1-to-1 Composition**. A User must have one and only one WalletIdentity, which cannot exist independently.
2. **User ↔ ItemListing: 1-to-Many**. One User (as seller) can create many ItemListings. One ItemListing is owned by exactly one User.
3. **User ↔ Transaction (as Buyer/Seller): 1-to-Many**. A User can be involved in many Transactions as either party. Each Transaction requires exactly one Buyer and one Seller (distinct Users).
4. **ItemListing ↔ Transaction: 1-to-0..1**. An ItemListing can have either zero (if unsold) or one (if sold) associated Transaction.
5. **User ↔ Message: 1-to-Many (Sender), 1-to-Many (Receiver)**. A User can send and receive many Messages.
6. **Transaction ↔ DigitalSignature: 1-to-1 Aggregation**. A Transaction aggregates one DigitalSignature (the buyer's), but the signature object has a conceptual identity of its own.

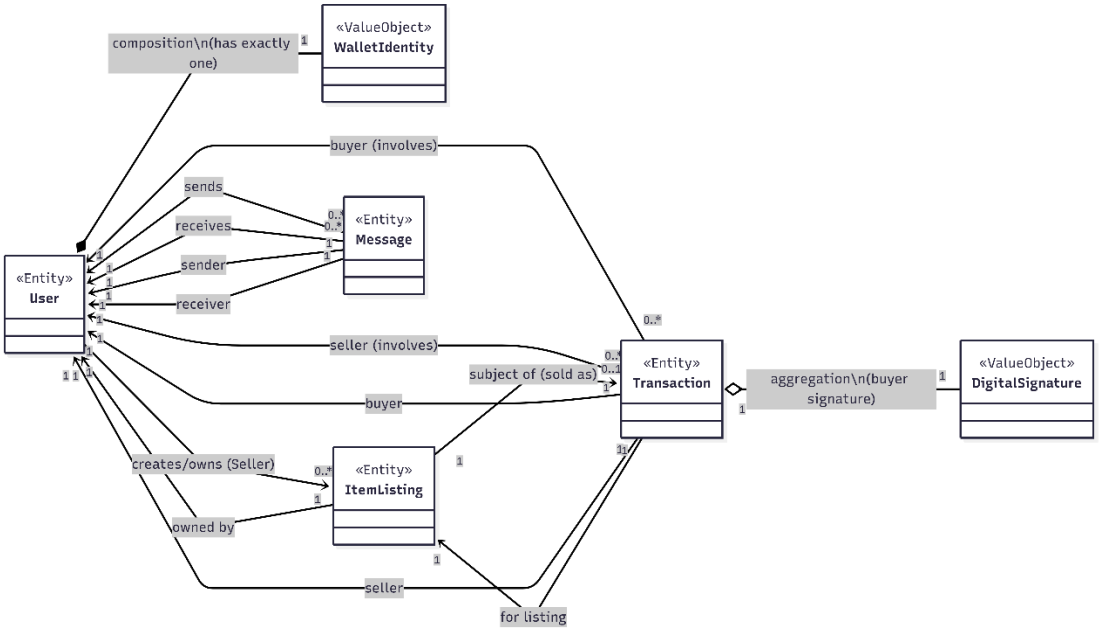


Fig. 3.3 Class Relationships and Multiplicity

## 3.4 Design Patterns Applied

**Aggregate Root (User, ItemListing, Transaction)**: These classes serve as the entry points for manipulating a cluster of related objects (e.g., a User and its WalletIdentity), ensuring transactional consistency.
**Value Object (WalletIdentity, DigitalSignature)**: These objects are defined solely by their attributes and have no independent lifecycle. Equality is based on attribute values.
**Repository Pattern (Implied)**: While not shown as a class, the persistence layer will implement Repository interfaces (e.g., UserRepository, ItemListingRepository) to abstract data access for these domain entities, aligning with the layered architecture.

# 4. Sequence Diagrams

## 4.1 User Authentication and Campus Verification Flow

Shows the complete process from wallet connection through campus email verification, including the Web3 challenge-response authentication mechanism.

1. Participants (Actors)
   - User: The student or staff member attempting to access the platform.
2. System Components (Objects)
   - Frontend: The React-based user interface running in the browser, responsible for wallet interactions and displaying login forms.
   - Wallet: The simulated Web3 wallet service (using ethers.js) responsible for generating keys and signing challenges.

- Backend: The Node.js/Express server responsible for authentication logic, session management, and email verification.
- Database: The PostgreSQL database responsible for storing user accounts and verification statuses.

3. Messages (Messages)
   - Initiate Login: User triggers the login process.
   - Generate Keys: Request to generate wallet key pairs.
   - Return Wallet Address: Return the public wallet address.
   - Request Challenge: Request a unique authentication challenge (nonce).
   - Generate/Store Challenge: Create and store a unique nonce in the database.
   - Return Challenge: Send the challenge string to the frontend.
   - Sign Challenge: Request to cryptographically sign the challenge.
   - Return Signature: Return the digital signature.
   - Verify Signature: Request to validate the signature against the wallet address and challenge.
   - Verify Signature Result: Confirmation that the signature is valid.
   - Submit Campus Email: User submits their university email for verification.
   - Send Verification Email: System sends a verification link to the user's email.
   - Update Verification Status: Update the user's record to "Verified".
   - Return Success: Final confirmation of successful login and verification.

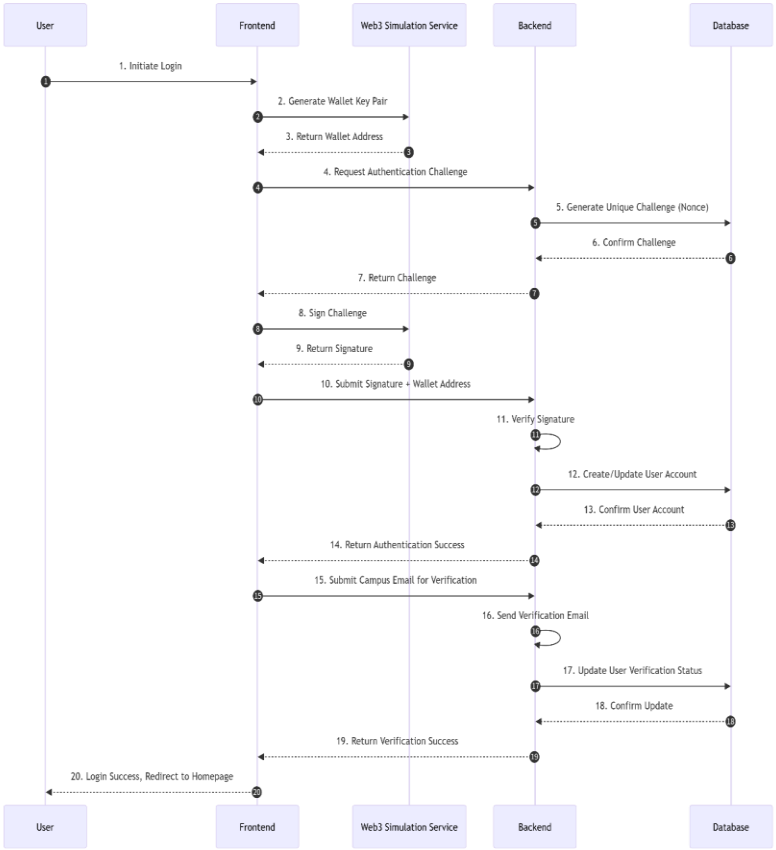


Fig.4.1 User Authentication and Campus Verification Flow

4. Control Flow
   - User initiates the login process on the Frontend.
   - Frontend requests the Wallet to generate a key pair.
   - Wallet returns the wallet address to the Frontend.
   - Frontend requests a login challenge from the Backend.
   - Backend generates a unique nonce, stores it in the Database, and returns it to the Frontend.
   - Frontend sends the challenge to the Wallet to be signed.
   - Wallet returns the digital signature to the Frontend.
   - Frontend sends the signature and wallet address to the Backend.
   - Backend verifies the signature validity (matches wallet address and challenge).
   - User submits their campus email address via the Frontend.
   - Backend sends a verification email and updates the user's status in the Database.
   - Backend returns a success response to the Frontend.

## 4.2 Item Listing Creation Flow

Demonstrates how a verified seller creates a new item listing, including input validation, backend verification, and database persistence.

1. Participants (Actors)
   - Seller: A verified user attempting to post a new item for sale.
2. System Components (Objects)
   - Frontend: The user interface where the seller fills in the item details (title, description, price, image).
   - Backend: The server API that handles listing requests and validates data.
   - Database: The database that stores the item records persistently.
3. Messages (Messages)
   - Create Listing: User submits the item creation form.
   - Validate Data: Check if required fields (title, price, description) are present.
   - Check Verification: Verify that the user has completed campus email verification.
   - Create Item Record: Insert the new item data into the database.
   - Confirm Creation: Return the Item ID and success message.
   - Display Success: Show confirmation to the user.
4. Control Flow
   - Seller fills in the item details (Title, Description, Price, Image) on the Frontend.
   - Frontend sends a POST /api/items/create request to the Backend.
   - Backend validates the input data to ensure all mandatory fields are included.
   - Backend checks if the Seller is verified (campus email verified).
   - Backend creates a new item record associated with the Seller's ID in the Database.
   - Database confirms the record insertion.
   - Backend returns a success response (including the new Item ID) to the Frontend.
   - Frontend displays a success message to the Seller.

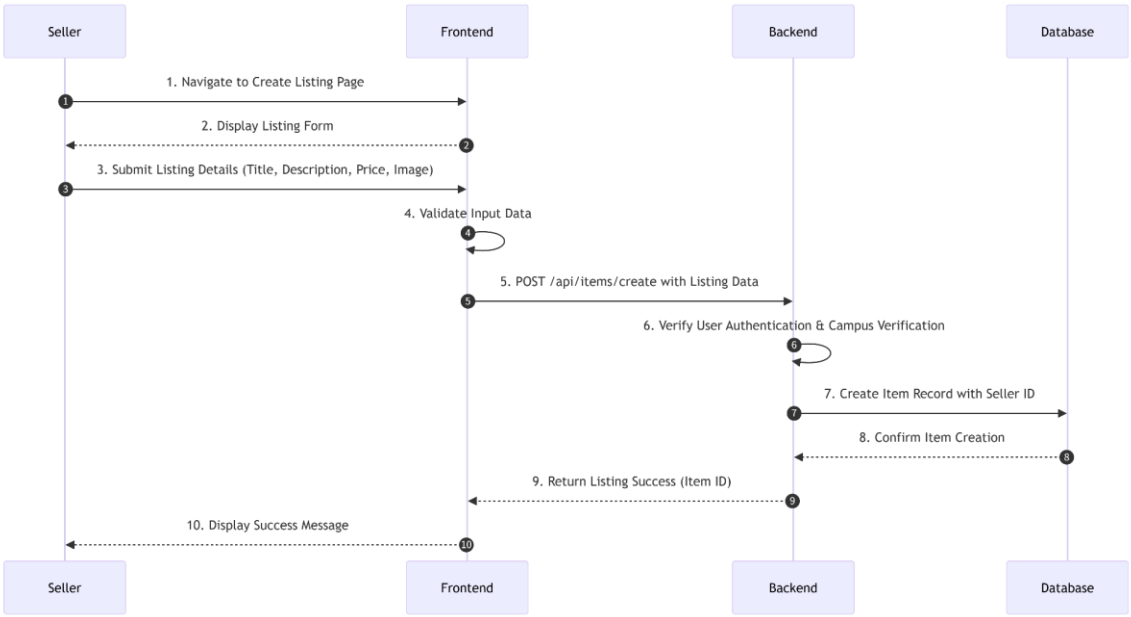


Fig.4.2 Item Listing Creation Flow

## 4.3 Simulated Transaction and Signing Flow

Illustrates the Web3 transaction simulation process where a buyer purchases an item, including transaction payload generation, digital signing, and status confirmation.

1. Participants (Actors)
   - Buyer: A verified user purchasing an item.
2. System Components (Objects)
   - Frontend: The product detail page and the simulated wallet interface.
   - Backend: The server handling transaction logic and payload generation.
   - Database: The database storing transaction records.
   - Wallet: The simulated wallet component used for signing the transaction.
3. Messages (Messages)
   - Click “Buy Now”: Buyer initiates the purchase.
   - Request Payload: Request to generate the transaction details (Item ID, Price, Addresses).
   - Retrieve Item/Seller Info: Fetch details from the database.
   - Return Payload: Send the structured transaction data to the frontend.
   - Display Transaction: Show the data to the buyer for confirmation.
   - Confirm Transaction: Buyer agrees to the terms.
   - Request Signature: Frontend asks the Wallet to sign the transaction data.
   - Return Signature: Wallet returns the digital signature.
   - Verify & Record: Backend verifies the signature and saves the transaction.
   - Update Status: Change transaction status to “Confirmed”.

- Return Success: Notify the frontend that the transaction is complete.

4. Control Flow
   - Buyer clicks the “Buy Now” button on the product details page in the Frontend.
   - Frontend requests a transaction payload from the Backend.
   - Backend retrieves Item and Seller information from the Database.
   - Backend constructs a transaction payload (containing Item ID, Price, Buyer Address, Seller Address).
   - Backend returns the payload to the Frontend.
   - Frontend displays the transaction details to the Buyer.
   - Buyer confirms the transaction details.
   - Frontend sends the payload to the Wallet to be signed.
   - Wallet returns the digital signature to the Frontend.
   - Frontend sends the signed payload to the Backend (POST /api/transactions/verify-signature).
   - Backend verifies the signature and updates the transaction status to “Confirmed” in the Database.
   - Backend returns a success message to the Frontend.
   - Frontend displays the transaction success message to the Buyer.

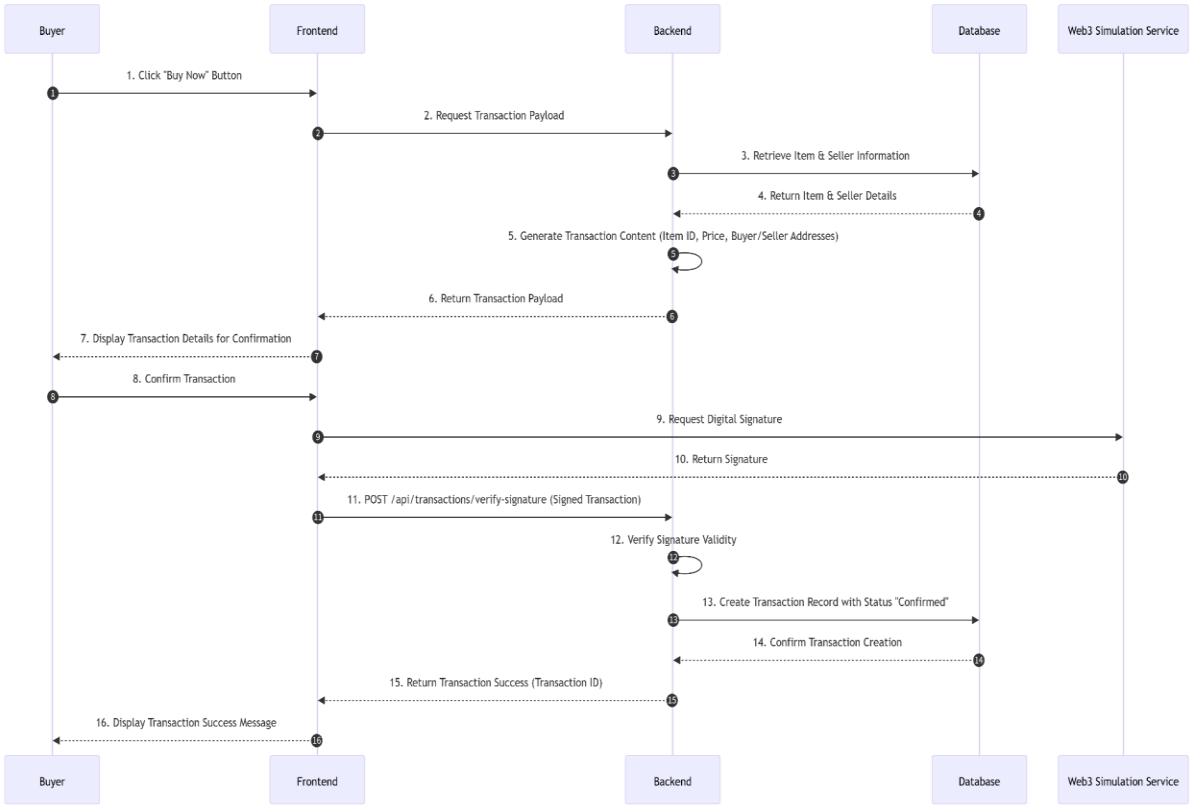


Fig.4.3 Simulated Transaction and Signing Flow

# 5. Dialog Map

The Dialog Map of Web3 Campus Marketplace covers interactions between users and the system, as well as between different user roles, focusing on core business scenarios such as user authentication, commodity listing management, transaction execution, real-time communication, and administrator supervision. The design follows the principles of "aligning scenarios with functional requirements, standardizing interaction processes, highlighting Web3 technical characteristics, and conforming to campus service attributes" to ensure that the interaction experience is consistent with the system's technical architecture and user needs, providing a clear basis for subsequent development, testing, and user experience optimization.

## 5.1. Design Goals and Principles

### 5.1.1 Core Objectives

1. Unify the system interaction language and process to ensure consistency and predictability of user experience across different modules.

2. Clarify the logical connection between each dialogue stage and functional requirements to avoid interaction gaps or logical conflicts.

3. Provide a practical interaction framework for the testing phase to facilitate the derivation of interaction test cases and verify the completeness of functions.

4. Highlight the core features of the project through dialogue design, including simulating Web3 wallet authentication, decentralized transaction signatures, campus identity verification, etc.

5. Optimize dialogue prompts and operation guidance to reduce user cognitive costs and enhance the accessibility and ease of use of the platform.

## 5.2 Dialog Map for Modules

### 5.2.1 User Authentication and Campus Verification Module

This module corresponds to the system's core requirement of "securing Web3-based authentication and campus verification", focusing on the dialogue logic of simulated Web3 wallet login, campus email verification, and permission control for unverified users.

**Initiator:** Guest/Unverified User; Trigger Condition: First visit to the platform and click the "Login" button
**System → User:** "Welcome to Web3 Campus Marketplace! To ensure transaction security, please connect your simulated Web3 wallet to complete identity authentication first."
**User Operation:** Click "Connect Wallet" (activate the simulated wallet)
**System Follow-up Response:** "Wallet connected successfully! Please sign the verification challenge to confirm your identity"

**Initiator:** User; Trigger Condition: After completing wallet connection
**System → User:** "Please sign the following verification code to confirm wallet ownership: [32-bit random Nonce value]"
**User Operation:** Click "Confirm Signature" (complete simulated signature)
**System Follow-up Response:** "Wallet authentication passed! Please enter your campus email ([e.ntu.edu.sg] suffix) to complete community verification"

**Initiator:** User; Trigger Condition: After submitting the campus email
**System → User:** "A verification link has been sent to [User's Email]. Please click the link to complete verification (valid for 24 hours)"
**User Operation:** Click the link in the email to complete verification
**System Follow-up Response:** "Campus identity verification successful! You have obtained full access to the platform"

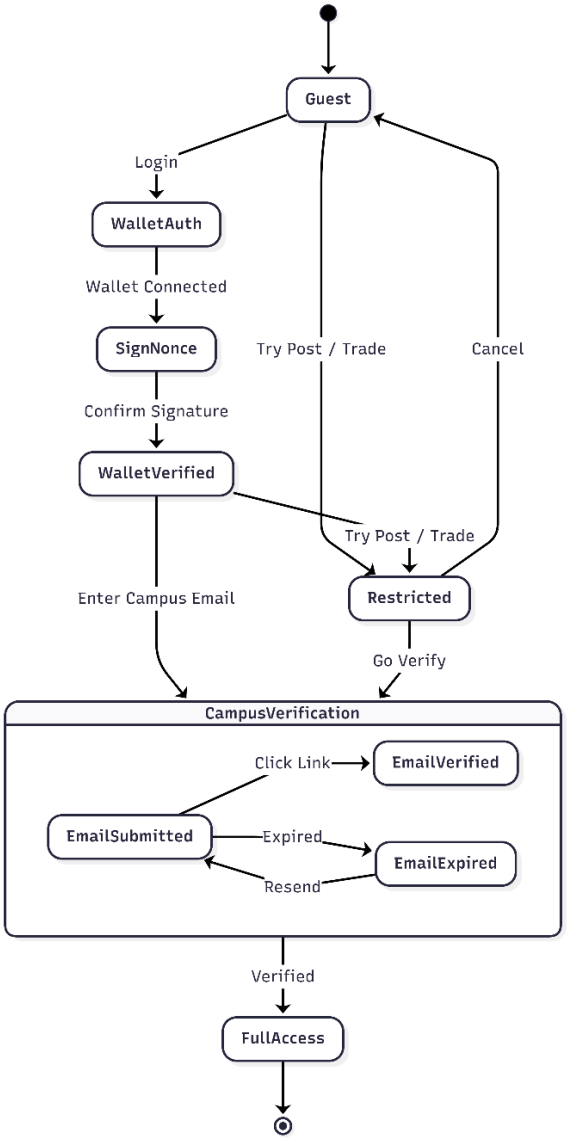


Fig.5.1 User Authentication and Campus Verification Dialog Map

**Initiator:** Unverified User; Trigger Condition: Attempt to post listings/initiate a transaction
**System → User:** "Sorry, you need to complete campus email verification first to use this function. Go to verify now?"
**User Operation:** Click "Go to Verify" or "Cancel"
**System Follow-up Response:** Jump to the verification page / Stay on the current page, and the function remains unavailable

**Initiator:** User; Trigger Condition: The email verification link expires/invalid
**System → User:** "The verification link is invalid or has expired. Please click to resend the verification email"
**User Operation:** Click "Resend Verification Email"
**System Follow-up Response:** "The verification email has been resent successfully"

### 5.2.2 User Profile and Account Management Module

This module focuses on the dialogue logic of user profile viewing, editing, account security management, and administrator's user management operations, ensuring the security and controllability of user account information.

**Initiator:** Authenticated User; Trigger Condition: Enter the "Personal Profile" page
**System → User:** Display profile information: "Wallet Address: [Masked Address] / Verification Status: Verified / Role: Student"
**User Operation:** Click "Edit Profile"
**System Follow-up Response:** Display editable fields: "Nickname, Avatar, Personal Signature" (Wallet address and verification status are non-editable)

**Initiator:** User; Trigger Condition: Submit edited profile (nickname contains sensitive words)
**System → User:** "The nickname contains non-compliant content. Please modify it and try again"
**User Operation:** Modify the nickname and resubmit
**System Follow-up Response:** "Profile updated successfully! The new information will take effect immediately"

**Initiator: User:** Trigger Condition: Click "View Account Security"
**System → User:** "Current Security Status: Safe / Binding Method: Simulated Web3 Wallet + Campus Email / Last Login Time: [Specific Time]"
**User Operation:** No operation or click "Modify Binding"
**System Follow-up Response:** Stay on the current page / "Wallet binding cannot be modified; campus email modification requires re-verification"

**Initiator: Administrator:** Trigger Condition: Search for a user in the backend
**System → Administrator:** Display user details: "Username: [Name] / Wallet Address: [Full Address] / Verification Status: [Status] / Registration Time: [Time] / Number of Listings Posted: [Number]"
**Administrator Operation:** Click "View Listings" or "Disable Account"
**System Follow-up Response:** Jump to the user's listings page / "Are you sure you want to disable this account? Disabling will take all their listings offline"

**Initiator: Administrator:** Trigger Condition: Confirm account disabling
**System → Administrator:** "Account [Username] has been disabled successfully. The user will receive a notification"
**System Follow-up Response:** Record the operation log and update the user's account status to "Disabled"

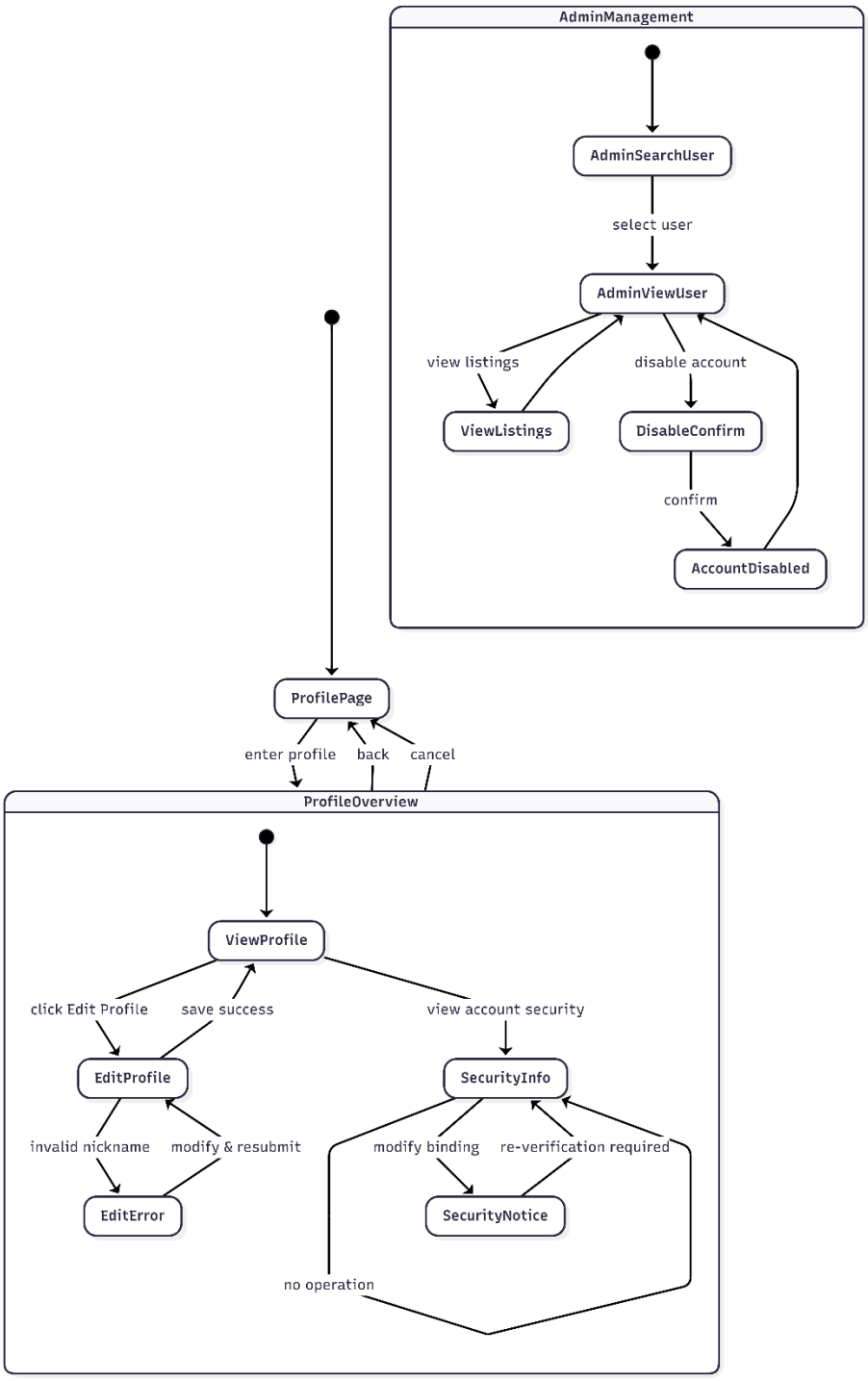


Fig.5.2 User Profile and Account Management Dialog Map

### 5.2.3 Item Listing Management Module

This module covers the dialogue logic of the entire lifecycle of item listings (creation, editing, deletion, status updates), ensuring that only verified users can perform listing operations and maintaining the standardization of listing information.

**Initiator:** Verified User; Trigger Condition: Click "Post New Listing"
**System → User:** Display the posting form: "Please fill in the product information (marked as required): Title*, Category*, Price*, Description*, Upload Images*"
**User Operation:** Fill in the information and submit (including images)
**System Follow-up Response:** "Listing submitted successfully! It will be displayed on the marketplace homepage after system review (usually within 10 minutes)"

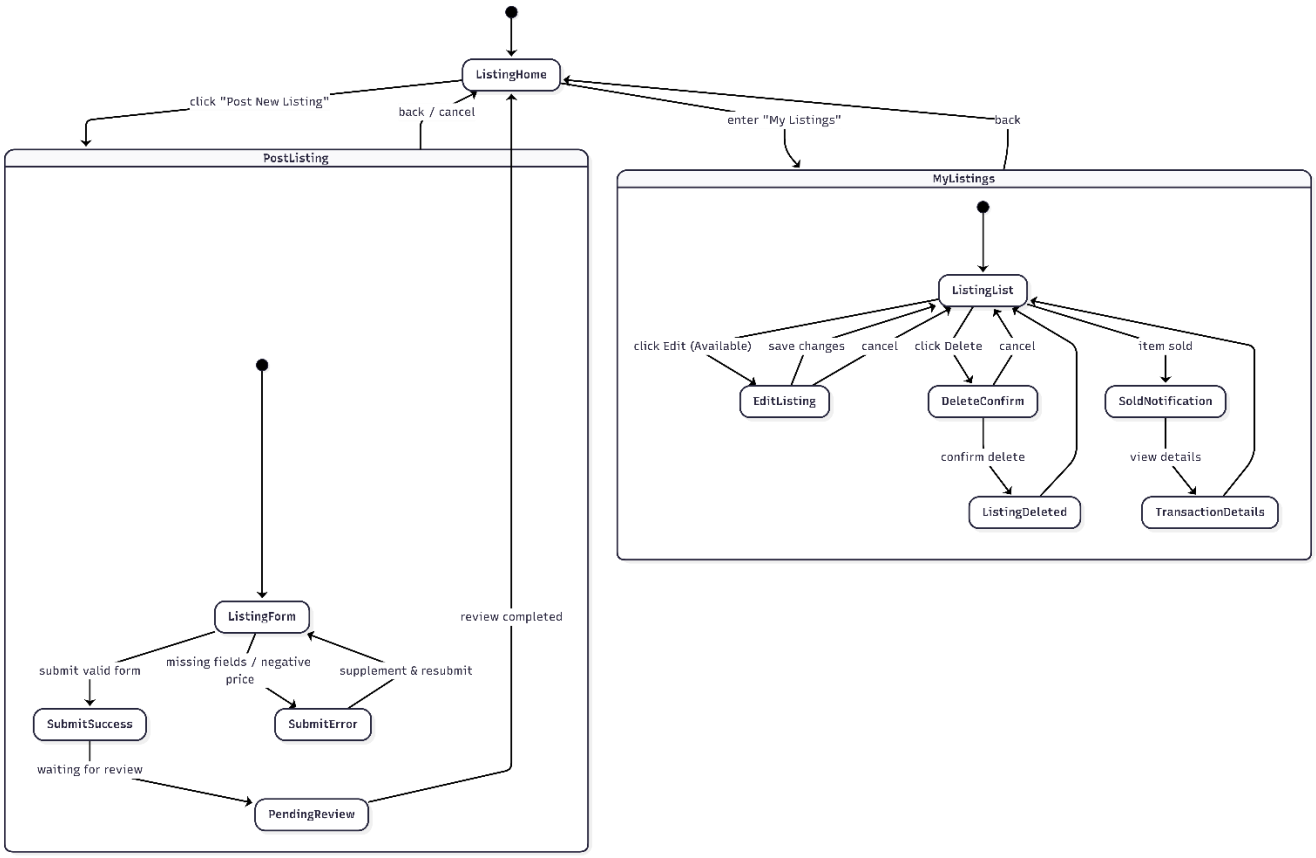


Fig.5.3 Item Listing Management Dialog Map

**Initiator:** User; Trigger Condition: Submit a listing with missing required fields/negative price
**System → User:** "Posting failed! Please check: 1. Title/Category/Price/Description not filled in; 2. Price cannot be negative"
**User Operation:** Supplement the information and resubmit
**System Follow-up Response:** "Listing submitted successfully! Waiting for review"

**Initiator:** Verified User; Trigger Condition: Enter "My Listings"
**System → User:** Display the listing list: "[Listing Title] / Price: [Amount] / Status: [Available/Sold/Pending Review] / Views: [Number]"
**User Operation:** Click the "Edit" button for an available listing
**System Follow-up Response:** Jump to the editing page with pre-filled original information: "Product information can be modified (Listing ID cannot be changed)"

**Initiator:** User; Trigger Condition: Click the "Delete" button for an available listing
**System → User:** "Are you sure you want to delete this listing? Deletion cannot be undone"
**User Operation:** Click "Confirm Delete" or "Cancel"
**System Follow-up Response:** "Listing deleted successfully" / Return to the listing list

**Initiator:** User; Trigger Condition: Listing transaction completed (product sold)
**System → Seller:** Push notification: "Your listing '[Title]' has been sold! You can view the details in 'My Transactions'"
**User Operation:** Click "View Details"
**System Follow-up Response:** Jump to the transaction details page