Problem 1

(a) Access Control Model Selection

Role-Based Access Control (RBAC)

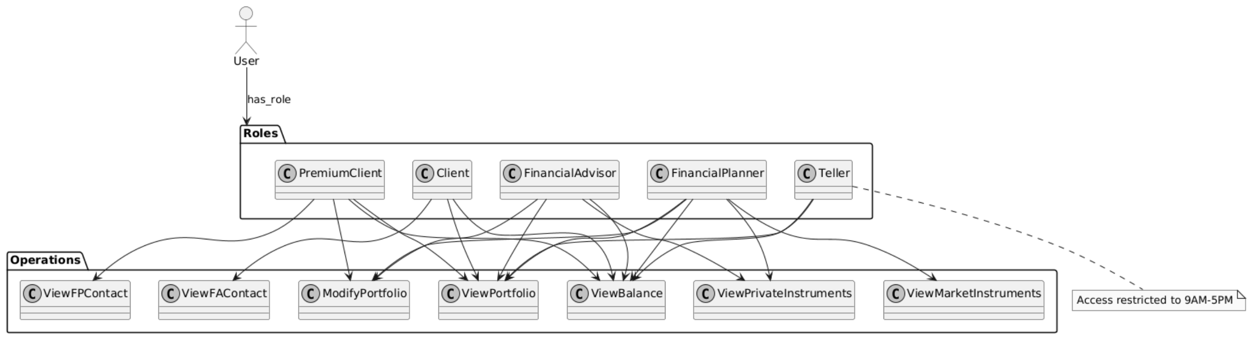
- RBAC help map the organizational structure with user roles (Client, Advisor, Teller, etc)

- Permissions are grouped according to the user's role

- Access requirements are organized by hierarchical structure, where employees have a set of base permissions, and additional role specific permissions.

- RBAC helps us manage role-based time restrictions

(b) Access Control Design Sketch



(c) Test Coverage:

- Role Permissions: Basic (Client) users, Premium Clients, Financial Advisors, Planners, Tellers)
- Time-Based Access Control: Teller only have access permissions during business hours)
- cross-role permission verification
- Input Validation: invalid role, operation, null input handling
- Edge Cases: role initialization, permission set completeness, isolation

Problem 2

(a) Hash Function Selection

PBKDF2-HMAC-SHA256

- **PBKDF2**: Specifically designed for password hashing
- **Hash Algorithm**: SHA-256 (Provides relatively cryptographic strong security for the minimal computation overhead)
- **Salt Length**: 32 bytes or 256 bits (The salt length should provide plenty of randomness to prevent pre-computation attacks)
- **Iterations**: 100,000 (Iteration count of 100,000 should be enough to protect from brute force attacks, while balancing performance)
- **Salt Generation**: os.urandom()

(b) Password File Structure

```
{
        "username": "verifyuser",
        "salt": "c3db22d5cf0ff25e3a617812080c0b45f9a7170f13f96aa3b010700db9a7dce4",
        "hash": "e6f12f6d8d1160a126b02e9d83844ad3013a8e12670d3dc407cf6f1491d06b76",
        "role": "Financial Advisor"
}
```

(c) Implementation Overview

add_user(username, password, role): Creates new user records

1. First validates that the input parameters are valid, not empty, etc
   - Prevents duplicate usernames
2. Generate a new salt of length 32 bytes for password hashing (unique for each user), creates secure password hash and converts the user data to JSON format before saving

verify_user(username, password): Authenticates users

3. Validates that the input parameters are valid and retrieves the stored user data as JSON from the password file.
4. Uses the stored salt to compute the password hash and compare it against stored. Returns authentication status.

(d) Test Coverage:

- File Management: File creation, custom paths, File integrity
- Password Security: unique salt, consistent hashing
- User Management: adding users, preventing duplicates, input validation
- Concurrent Access: multiple instances, file locking, data consistency

Problem 3

(a) User Interface Design

The implementation aims to only collect the necessary information:

- Username (unique identifier)
- Password (with proactive checking)
- Role selection (from predefined options)

Interface Flow:

```
=== justInvest User Enrollment ===

Enter username: [user input]

Available roles:

- Client

- Premium Client

- Financial Advisor

- Financial Planner

- Teller

Enter role: [user input]

Enter password: [user input]

Confirm password: [user input]
```

(b) Proactive Password Checker Implementation

Length Requirements:

- Minimum: 8 characters
- Maximum: 12 characters

Complexity Rules:

- At least one uppercase letter, one lowercase letter, one numerical digit
- One special character from: !, @, #, $, %, *, &

Security Measures:

- Prevents duplicate usernames
- Checks passwords are not a part of common passwords list
- Password handling is Case-sensitive
- Also checks is password is a part of weak password list

(c) Test Coverage:

- Password Validation: password length, meets complexity requirements
- Username Matching: case-insensitive, duplicates
- Enrollment Flow: Successful enrollment, duplicate prevention, role validation

Problem 4

(a) Login Interface Design

```
=== justInvest Login ===

Username: [user input]

Password: [user input]
```

(b) Access Privileges Display

```
=== User Information ===
Username: [authenticated username]
Role: [user role]


Access Privileges:
- [list of permitted operations]
```

(c) Test Coverage

- Authentication Tests: validating credentials
- Role-Based Permission Tests: for all roles
- Interface Testing: successful login, validating empty fields, invalid user
- Edge Cases: session management, multiple login attempts, permission display