
SC2006
TEAM 2
love

CONTENTS

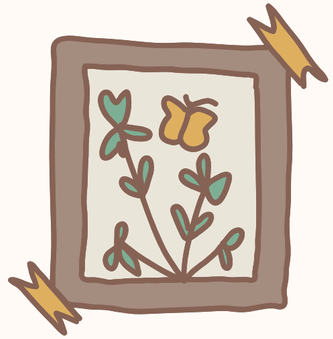
- PROBLEM STATEMENT
- USE CASE DIAGRAM
- EXTERNAL APIS
- GOOD SOFTWARE ENGINEERING PRACTICES
- TRACEABILITY

PROBLEM STATEMENT

No platform that facilitates the good redistribution to lower income families

Waste of resources

LEADING QUESTION



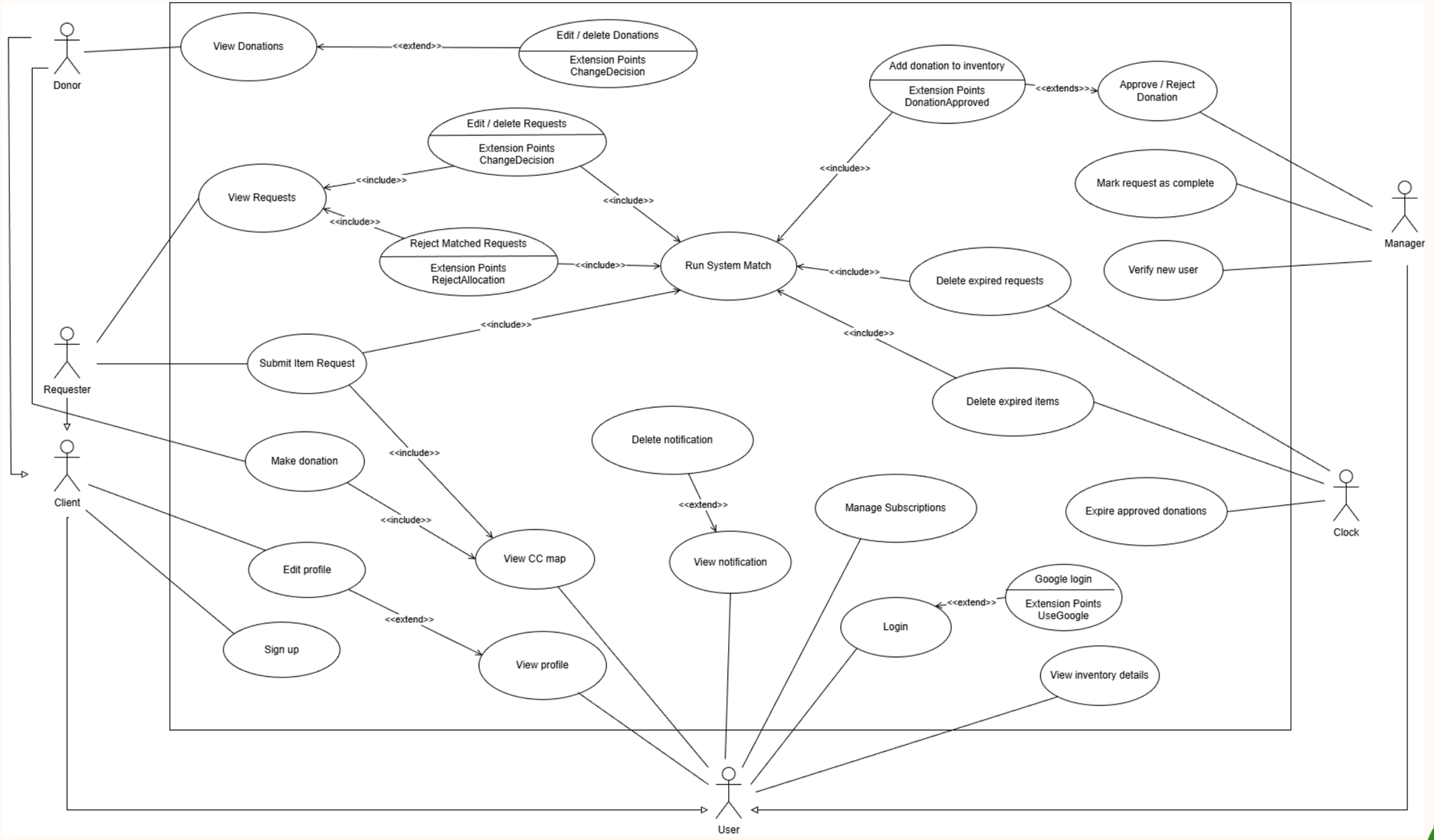
How might we create an accessible, community-driven platform that efficiently matches surplus goods from donors to the real-time needs of lower-income families through local Community Clubs?

OUR SOLUTION

CareConnect

Connect the community through Tech and compassion!

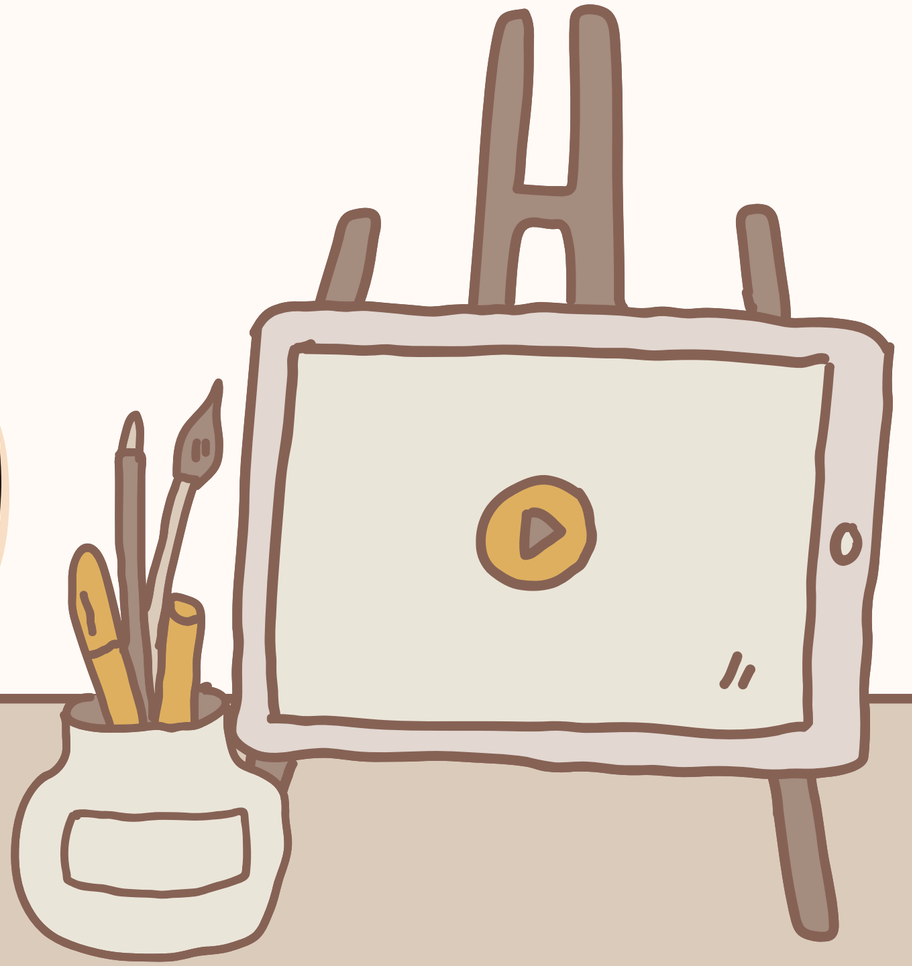
USE CASE DIAGRAM



EXTERNAL APIS

- Google OAuth 2.0 → Seemless authentication for users
- Supabase Storage → Store and manage images and files securely
- Community Club GeoJSON API → Provides the location and details of all Community Clubs in Singapore
- OneStreetMap API → Display and interact with community club locations on a dynamic map interface

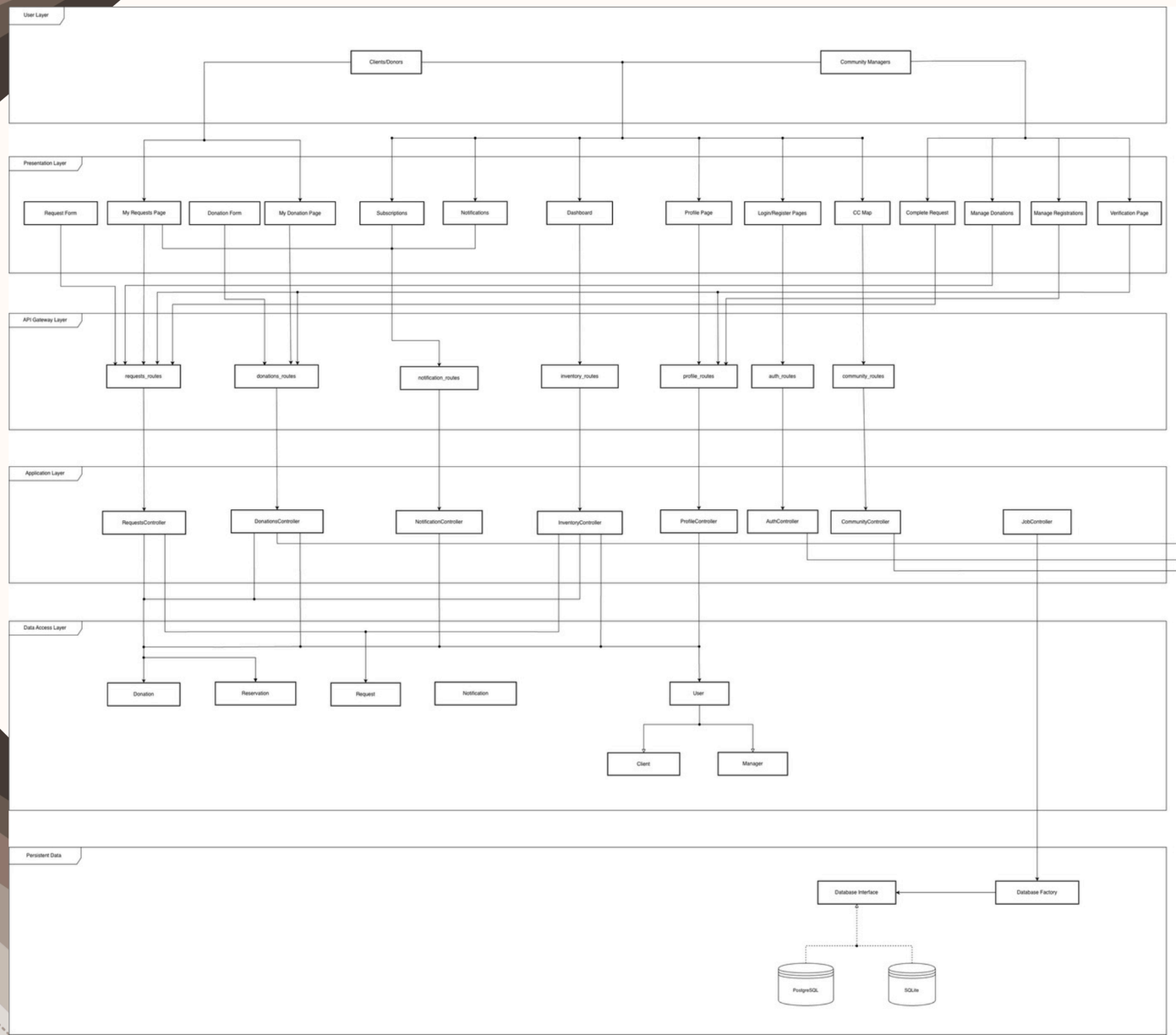
LIVE DEMO



GOOD SWE PRACTICES

- System Design
- Design Patterns Implementation
- Code Quality Practices

LAYERED ARCHITECTURE



User Layer

Presentation Layer

API Gateway

Application + External API

Data Access Layer

Persistent Data

DESIGN PATTERN

Strategy Pattern

Observer Pattern

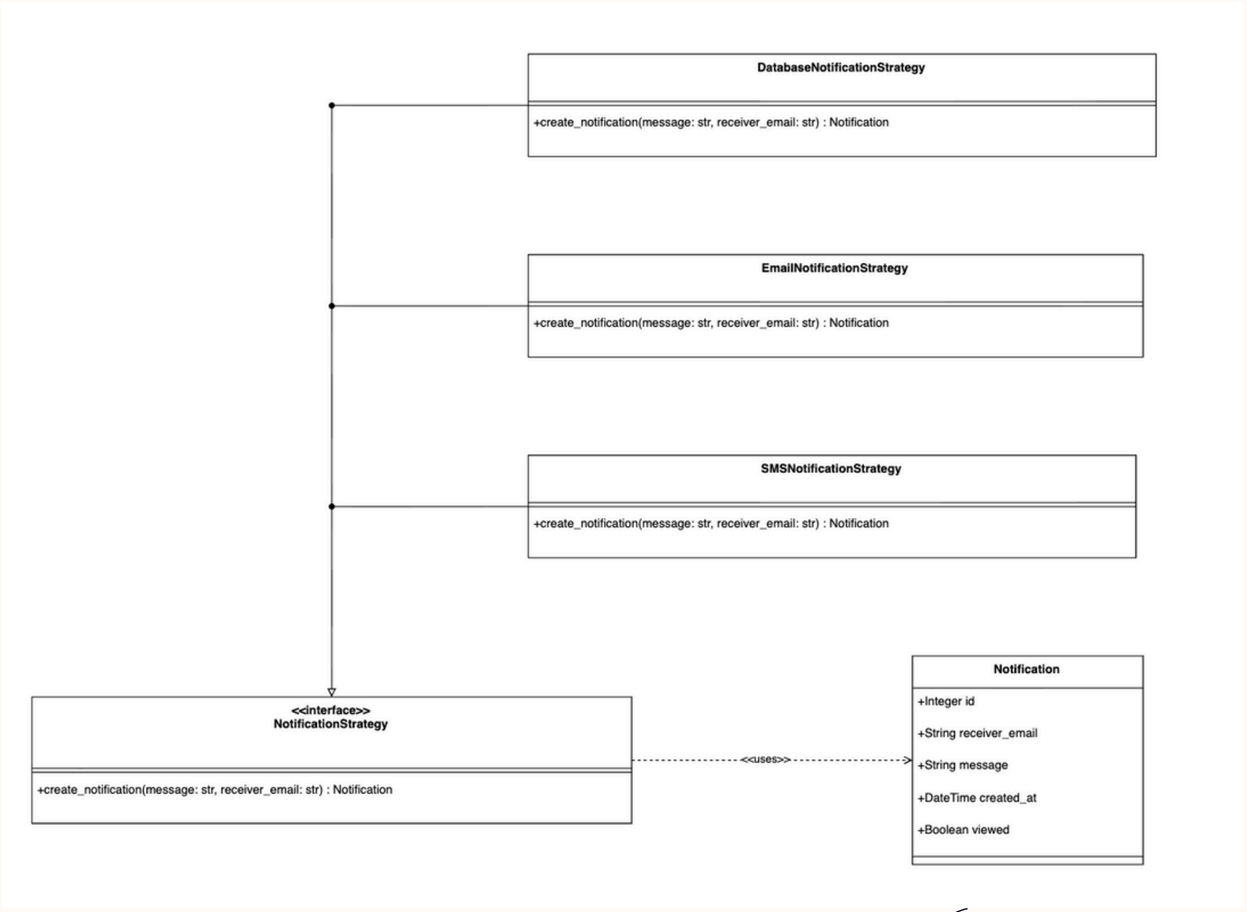
Factory Pattern

Facade Pattern

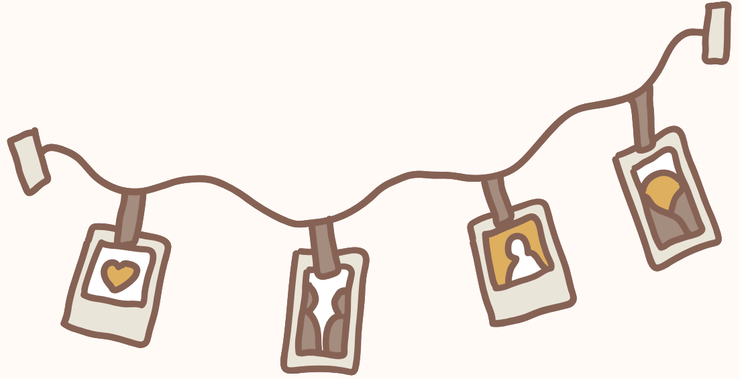
SOLID Principle

STRATEGY PATTERN

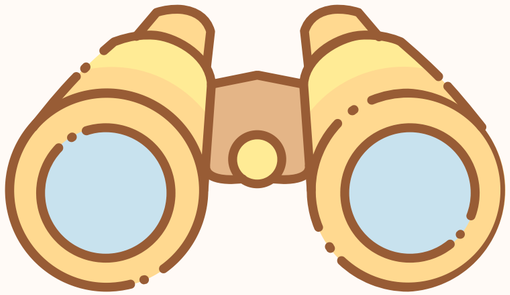
Authentication Strategy

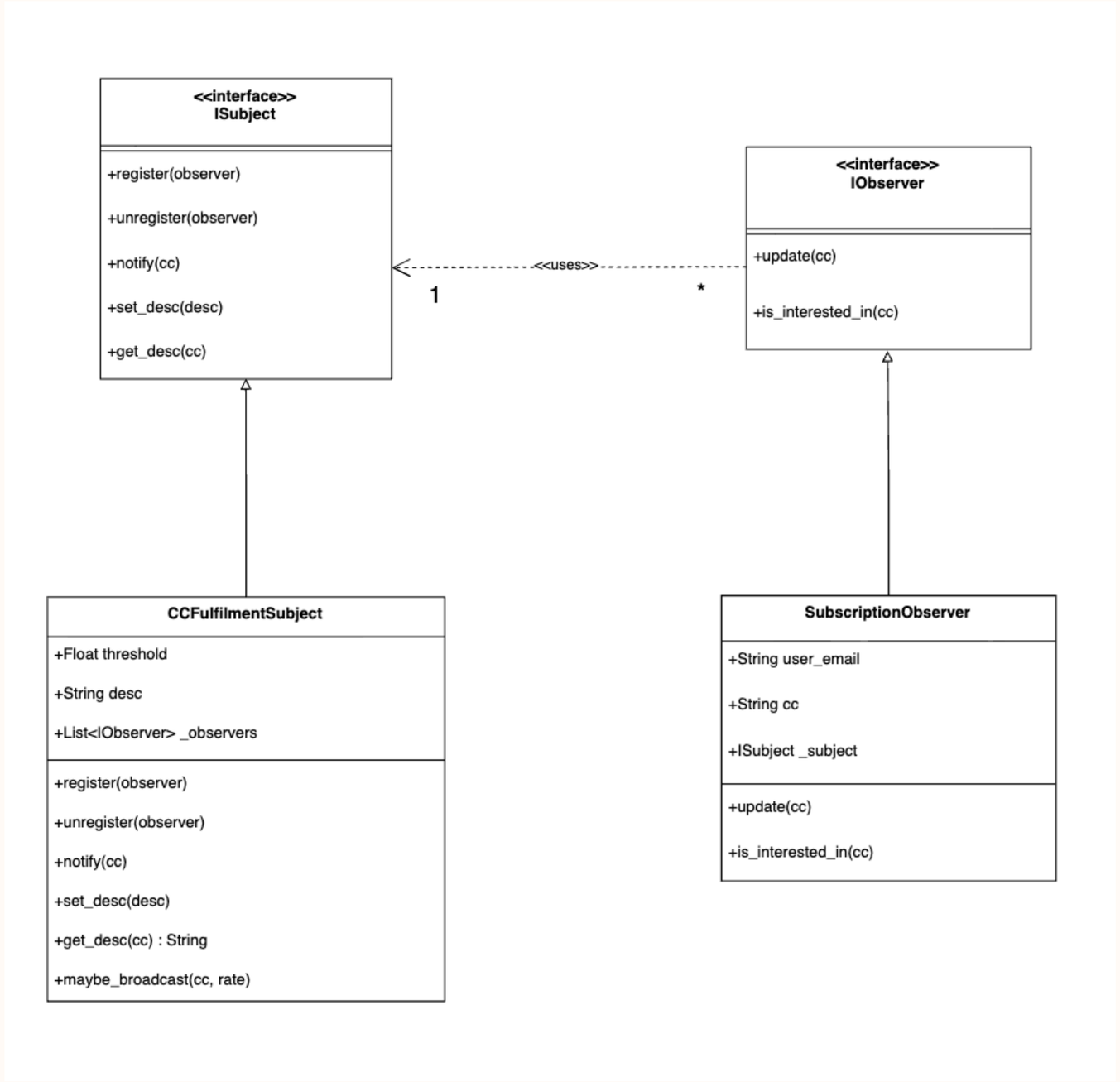


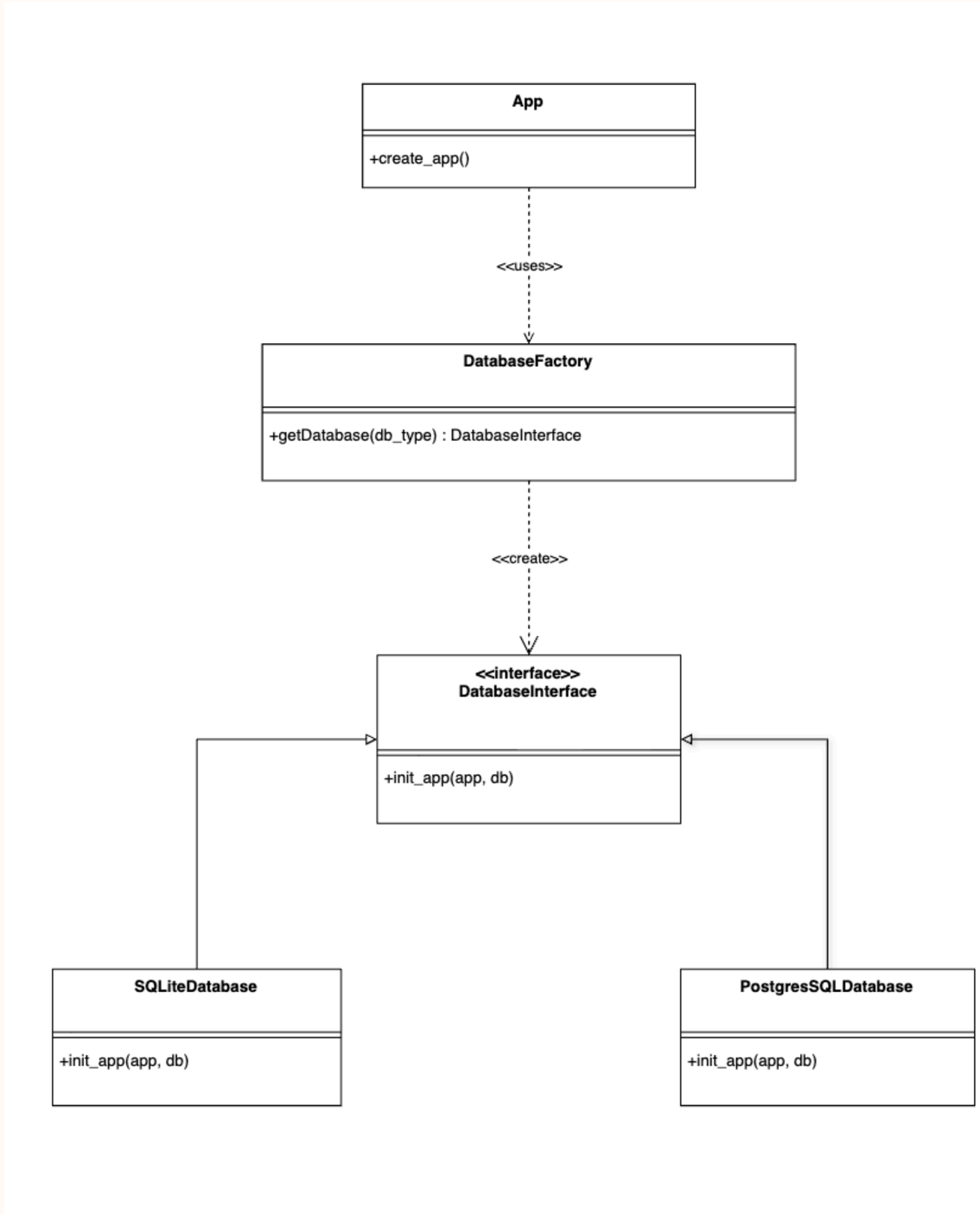
Notification Strategy



OBSERVER PATTERN







FACTORY PATTERN

FACADE PATTERN

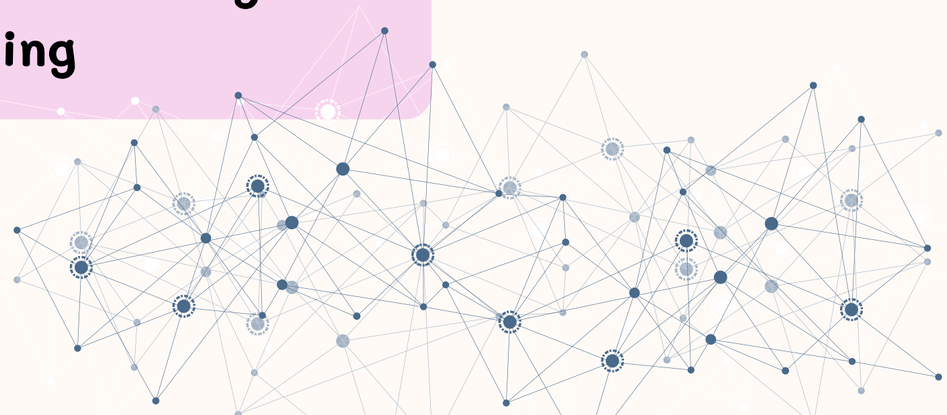
What's hidden?

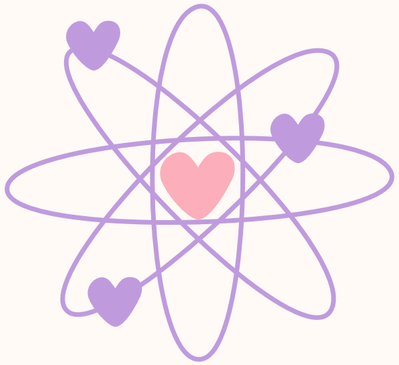
Controllers

- User authentication and role validation
- Database queries and updates
- Background jobs like allocation algorithm triggering
- Data response and error handling
- JSON response formatting

DatabaseFactory

- Connection set up between application and database interface (Flask + SQLAlchemy)
- Environment-specific settings





SOLID PRINCIPLE

Single Resposibility Principle

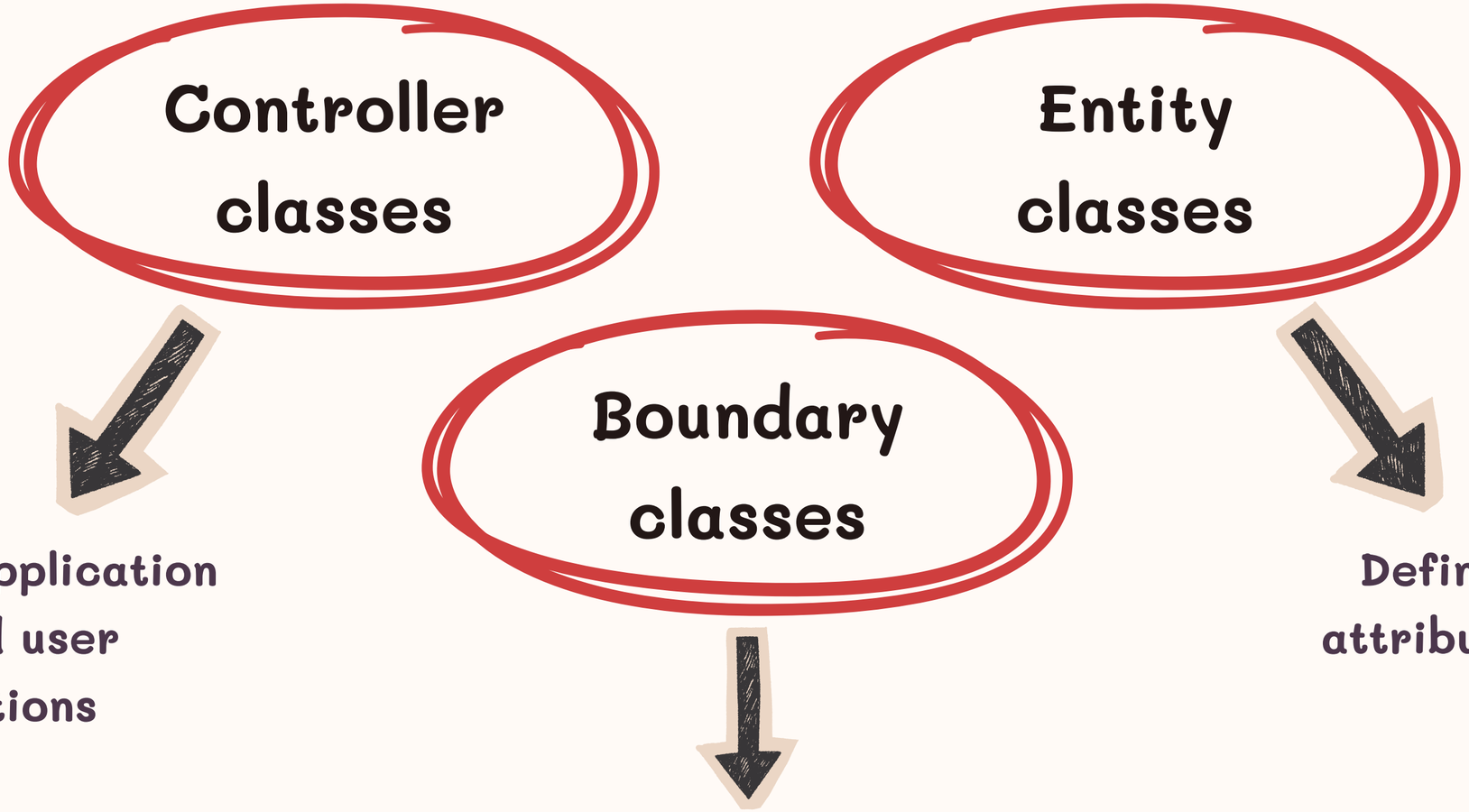
Open/Closed Principle

Liskov Substitution Principle

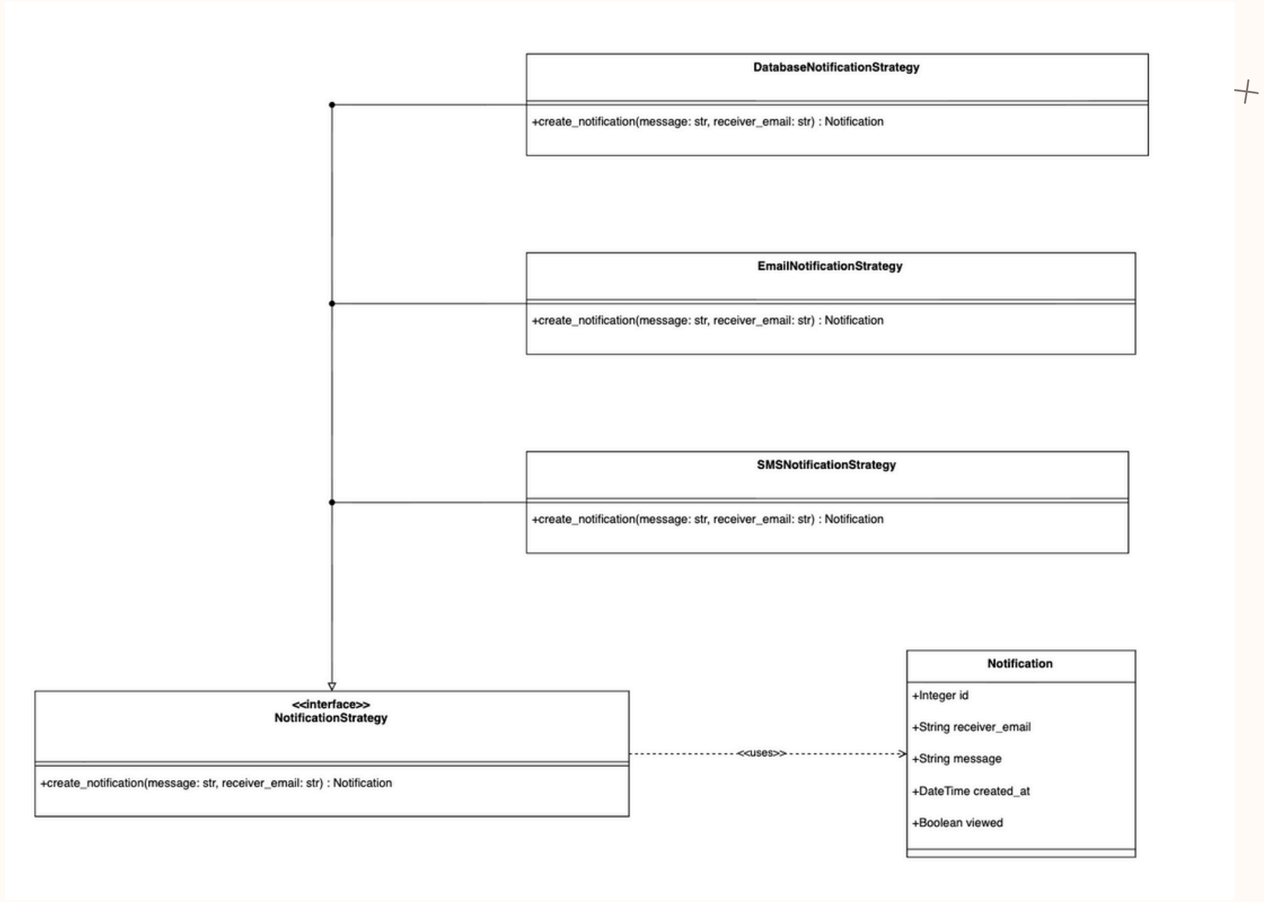
Interface Segregation Principle

Dependency Inversion Principle

SINGLE RESPINSIBILITY PRINCIPLE



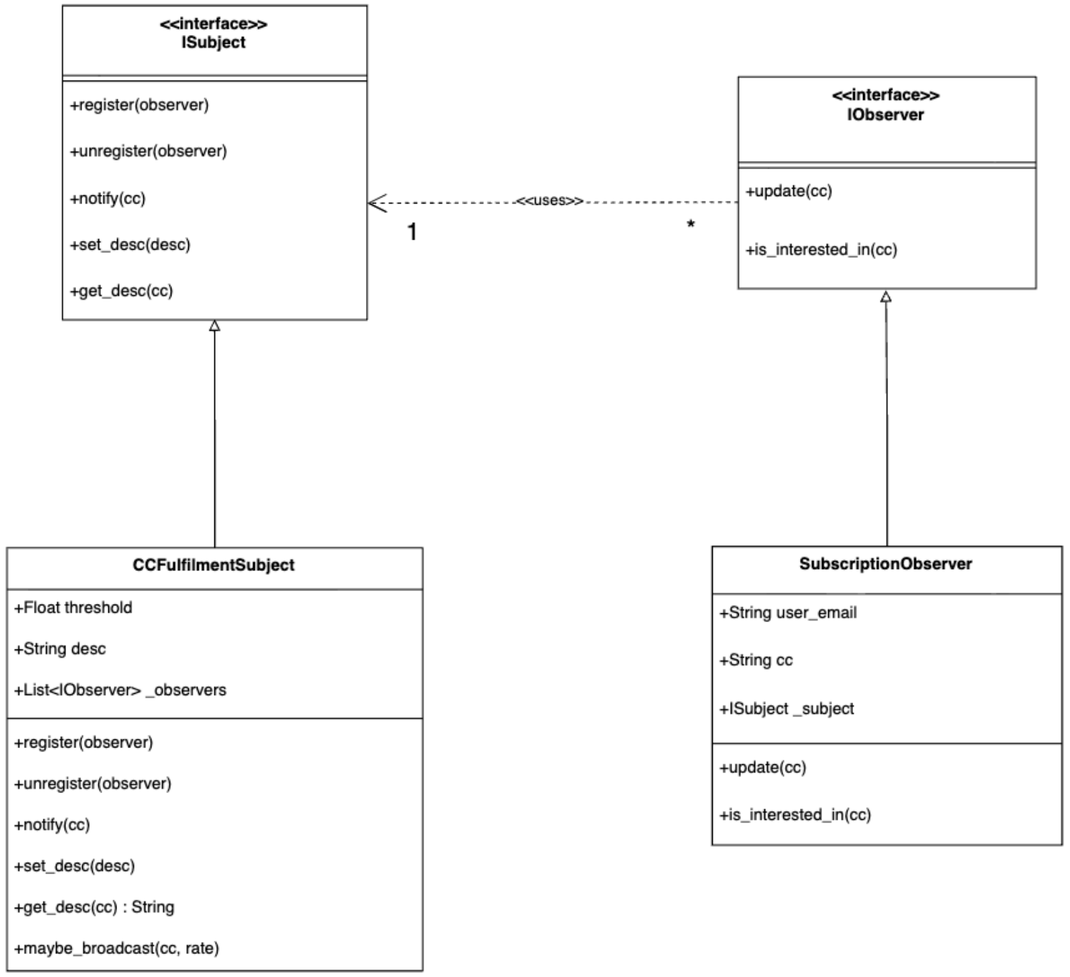
OPEN/CLOSED PRINCIPLE



Notification strategy

To add a new notification logic, you just create a new strategy class that implements the NotificationStrategy interface — no need to alter the main matching engine.

OPEN/CLOSED PRINCIPLE



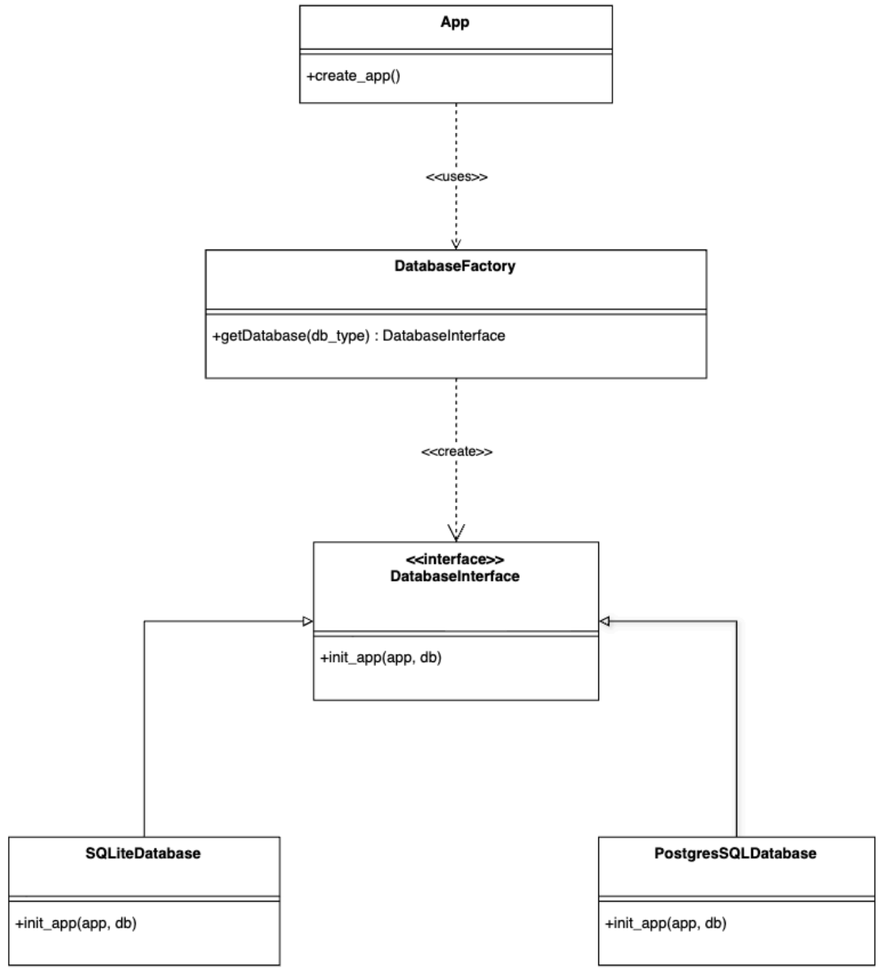
Broadcast observer

The observer interface allows new observer types to be added without changing the existing code

OPEN/CLOSED PRINCIPLE

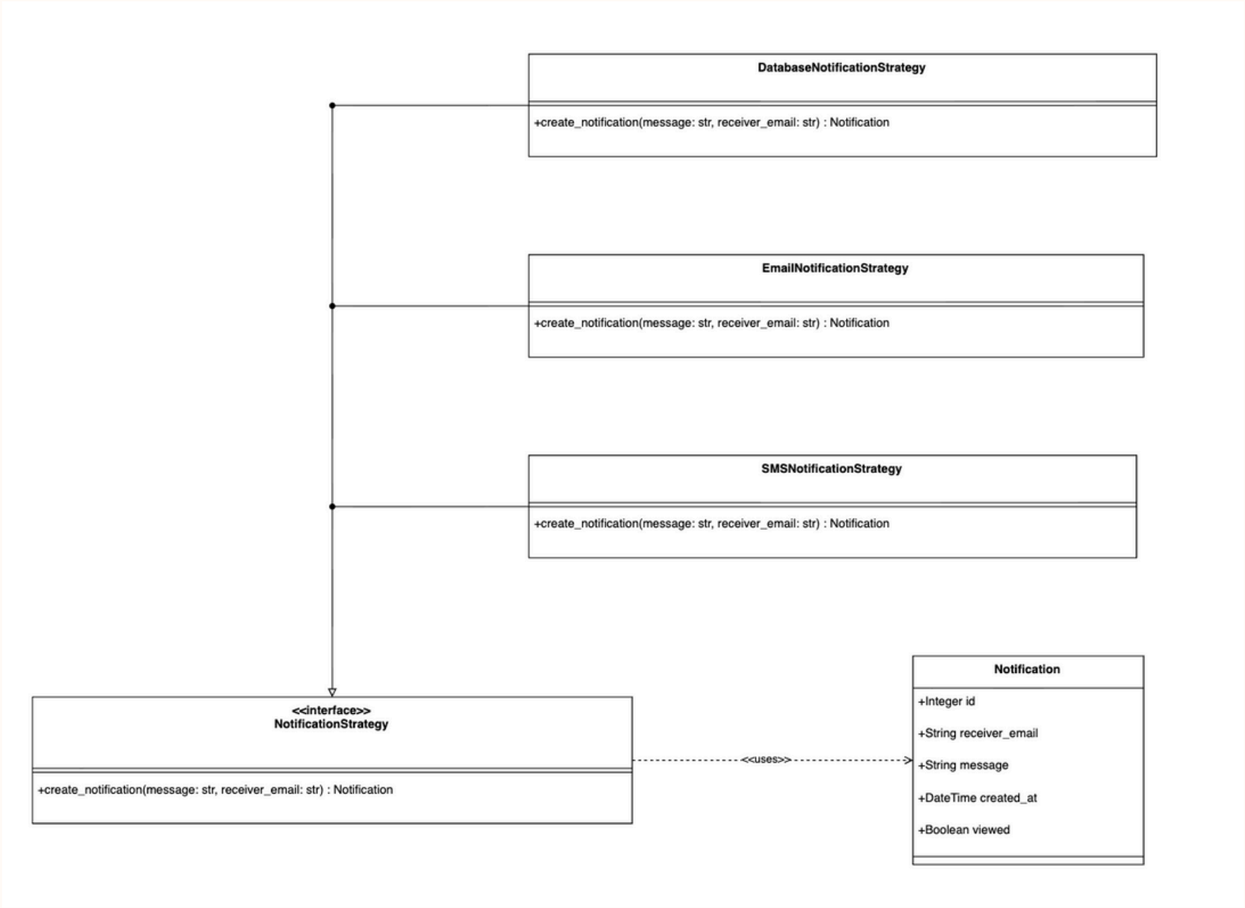
Database Factory

- The database factory can create different databases — SQLite, PosgreSQL
- A new database type is added as a subclass, not by editing existing logic.



LISKOV SUBSTITUTION PRINCIPLE

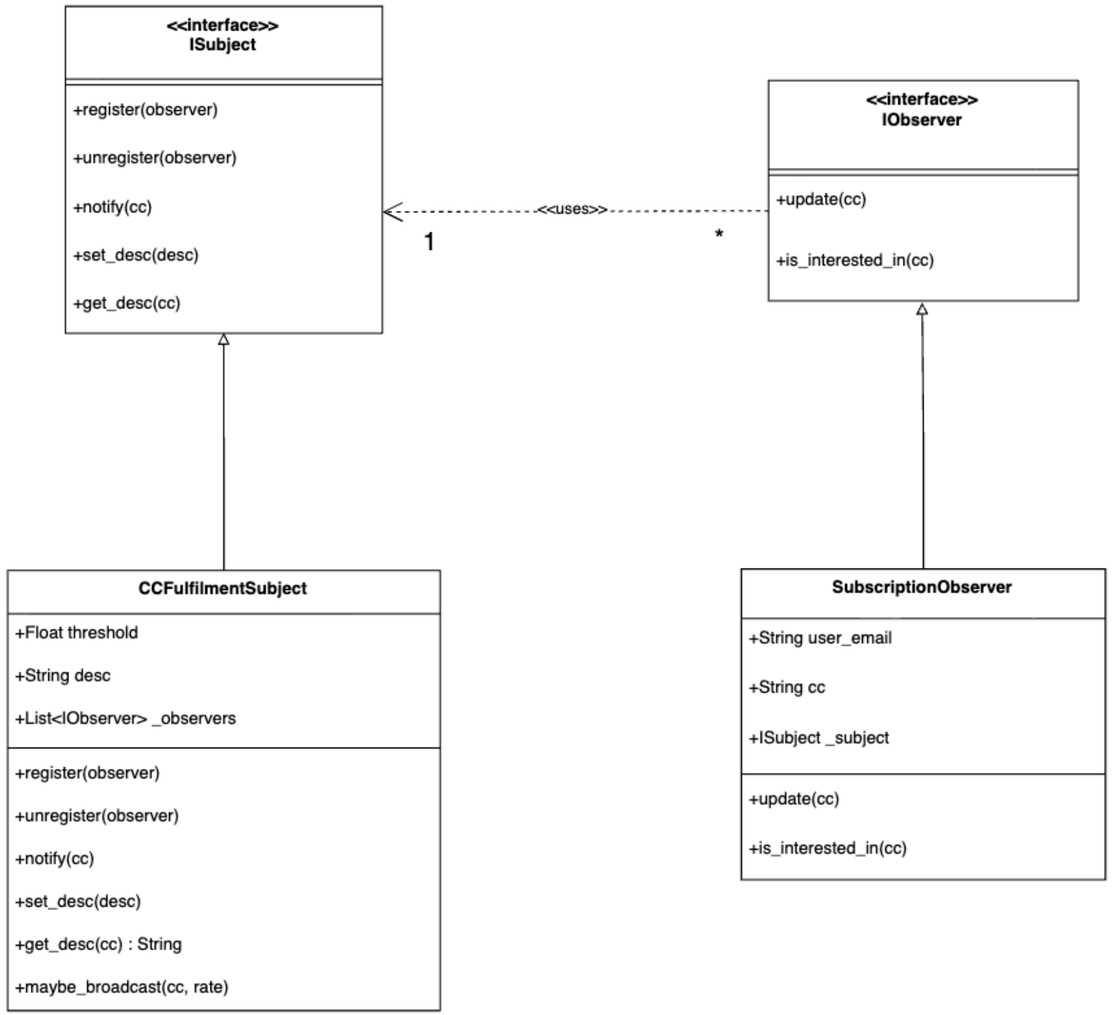
Eg. Notification strategy



The context object only interacts with the base type (NotificationStrategy)

Notification strategies can be swapped interchangeably

INTERFACE SEGREGATION PRICNCIPLE

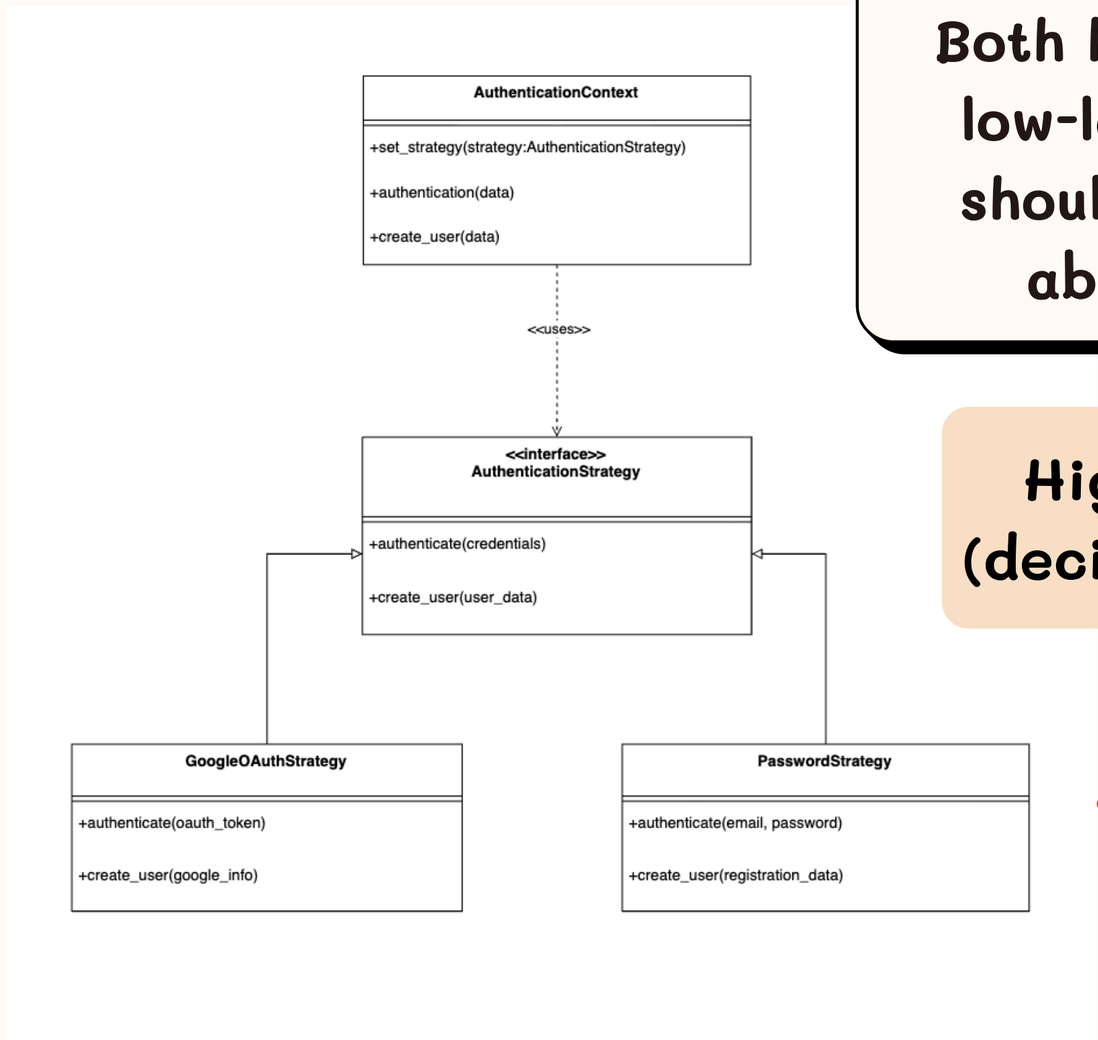


Focused Interfaces in Observer Pattern

ISubject **IObserver**

Classes only implement what they actually need → They do not need to handle unrelated methods

DEPENDENCY INVERSION PRINCIPLE



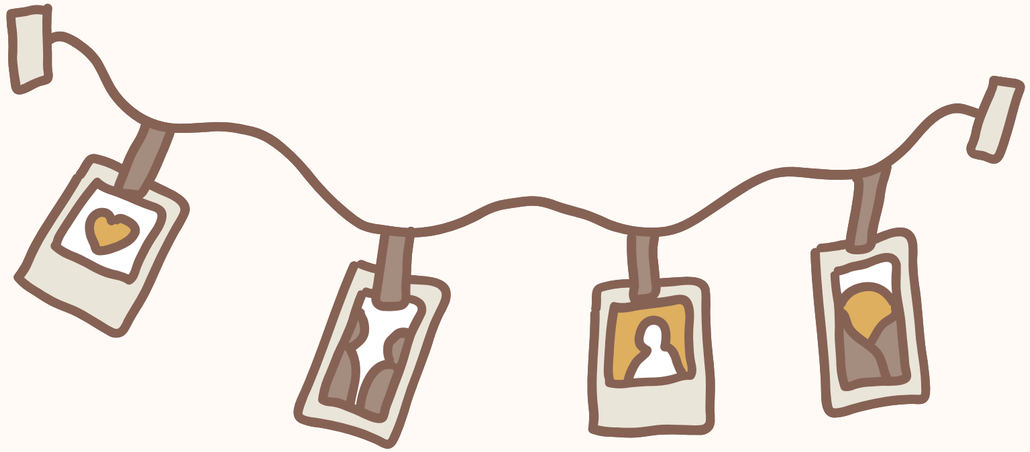
Both high-level and low-level modules should depend on abstractions

High-level module: Authentication context (decides what authentication strategy to use)

✗ **Depend on specific implementations i.e. GoogleOAuthStrategy/PasswordStrategy**

✓ **Easily plug in new strategies without changing the context**

CODE QUALITY PRACTICES



Error Handling

Reusability

Security

Documentation

CODE REUSABILITY

Cases that triggers the allocation process

CALLS →

run_allocation()

```python
def delete_pending_request(req_id: int):
    """Delete a pending request.

    Delete a user's own Pending request.
    If any items are (already) allocated/reserved, release them first.

    Args:
        req_id (int): ID of request to delete.

    Returns:
        tuple: JSON response and HTTP status code.
    """
```

```python
def manager_add(donation_id: int):
    """Add approved donation to inventory.

    Approve → Added:
    - Create Item rows as Available only (no reservations).
    - Allocation is performed by the 10-min background allocator.

    Args:
        donation_id (int): ID of donation to add to inventory.

    Returns:
        tuple: JSON response with created items and HTTP status code.
    """
```

```python
def create_request():
    """Create a new request.

    Returns:
        tuple: JSON response with request data and HTTP status code.
    """
```

```python
def run_expire_matched_requests_once(days_until_expire: int = 2) -> Dict[str, str]:
    """Expire old matched requests that haven't been collected.

    After N days in 'Matched', free the items and mark request as 'Expired'.

    Args:
        days_until_expire (int): Number of days before expiring matched requests.

    Returns:
        dict: Job execution results.
    """
```

```python
def run_cleanup_expired_items_once() -> Dict[str, object]:
    """Clean up expired donation items.

    Daily cleanup at Singapore midnight:
    - Find donations whose expiryDate is BEFORE today's Singapore date.
    - Remove ALL their items.
    - For any reservations referencing those items:
        * decrement the request's allocation,
        * if that request was 'Matched', set it to 'Pending' and clear matched_at

    Returns:
        dict: Job execution results with counts and affected items.
    """
```

```python
def reject_matched_request():
    """Reject a matched request.

    Returns:
        tuple: JSON response and HTTP status code.
    """
```

```python
def run_allocation() -> Dict[str, str]:
    """Match pending requests to available items using FIFO algorithm.

    Matches Pending requests to Available items (FIFO within each queue).
    Updates request status to 'Matched' when fully allocated and sends
    notifications to requesters.

    Returns:
        dict: Allocation job execution results.
    """
```

CODE REUSABILITY

Cases that triggers the broatcast notification check

CALLS

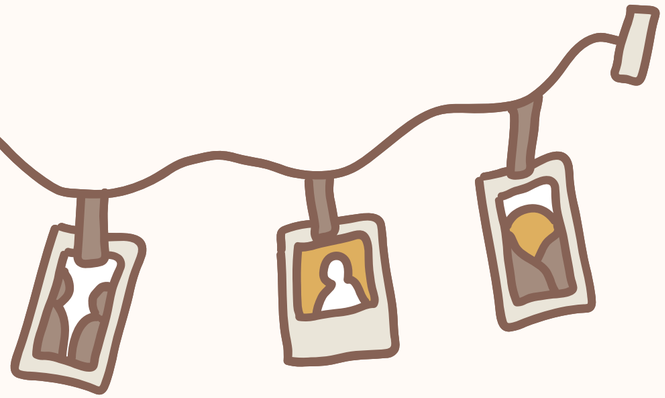
check_and_broadcast_for_cc(cc : str)

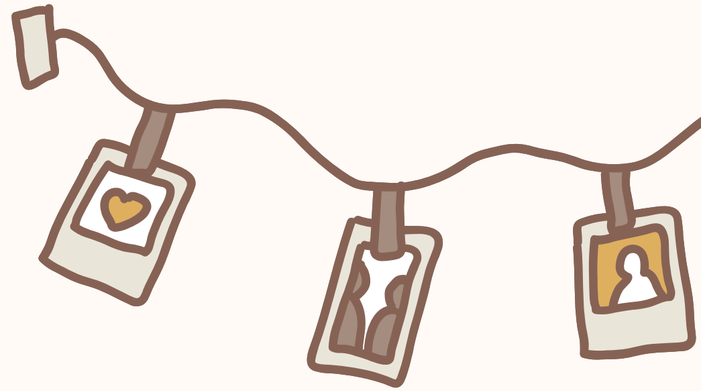
```python
def run_cleanup_expired_items_once() -> Dict[str, object]:
    """Clean up expired donation items.

    Daily cleanup at Singapore midnight:
    - Find donations whose expiryDate is BEFORE today's Singapore date.
    - Remove ALL their items.
    - For any reservations referencing those items:
        * decrement the request's allocation,
        * if that request was 'Matched', set it to 'Pending' and clear matched_at.

    Returns:
        dict: Job execution results with counts and affected items.
    """
```

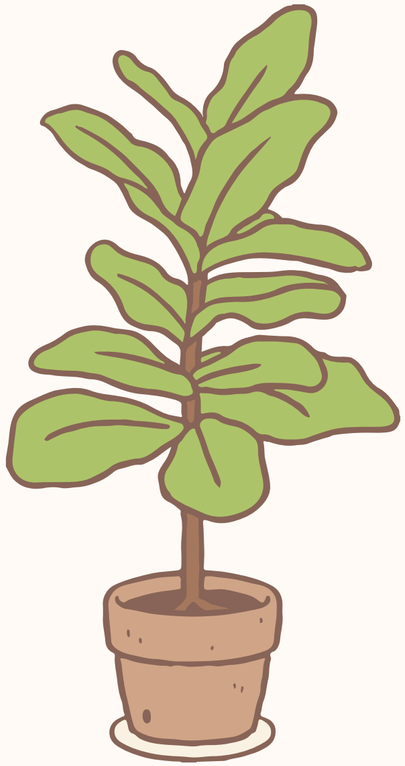
```python
def create_request():
    """Create a new request.

    Returns:
        tuple: JSON response with request data and HTTP status code.
    """
```

```python
def check_and_broadcast_for_cc(cc: str) -> float:
    """Check fulfillment rate for a community club and broadcast if low.

    Args:
        cc (str): Community club name.

    Returns:
        float: Fulfillment rate (0.0 to 1.0).
    """
```

CODE REUSABILITY







- Components
  - CommunityClubsMap.jsx
  - DonationCard.jsx
  - TopNav.jsx
- pages
  - ClientUI
  - MainUI
  - ManagerUI

Reused components in mutiple frontend displays

ERROR HANDLING

Try-Exception block with proper rollback mechanisms

```python
try:
    request_quantity = int(request_quantity)
    if request_quantity < 1: raise ValueError
except Exception:
    return jsonify({"message": "request_quantity must be a positive integer"}), 400
if not location: return jsonify({"message": "location (CC) is required"}), 400

try:
    req = Request(
        requester_email=u.email,
        request_category=request_category,
        request_item=request_item,
        request_quantity=request_quantity,
        location=location,
        status="Pending",
        allocation=0,
    )
    db.session.add(req)
    db.session.commit()

    run_allocation()

    # Check this CC's fulfilment after adding the new request
    check_and_broadcast_for_cc(location)

    return jsonify({
        "id": req.id, "status": req.status, "location": req.location,
        "request_item": req.request_item, "request_quantity": req.request_quantity,
        "allocation": req.allocation
    }), 201
except Exception as e:
    db.session.rollback()
    return jsonify({"message": "Failed to create request", "error": str(e)}), 500
```

1

Try Block

Contains code that might fail (e.g., updating database, division by zero).

Error occurs!

2

Except Block

Runs to handle the specific error

Program does not crash if there is an error

DOCUMENTATION

Quick Start

Prerequisites

- Python 3.8+
- Node.js 16+
- PostgreSQL (for production) or SQLite (for development)
- Redis (for session management)

Backend Setup

1. Clone and navigate to backend

```
cd backend
```

2. Create virtual environment

```
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. Install dependencies

```
pip install -r requirements.txt
```

4. Environment configuration

```
cp .env.example .env
# Edit .env with your configuration
```

5. Run the application

```
python -m backend.app
```

Frontend Setup

1. Navigate to frontend

```
cd frontend
```

2. Install dependencies

```
npm install
```

3. Environment configuration

```
cp .env.example .env
# Configure API endpoints
```

4. Start development server

```
npm run dev
```

SET-UP GUIDE

Guide future developers

Project Structure

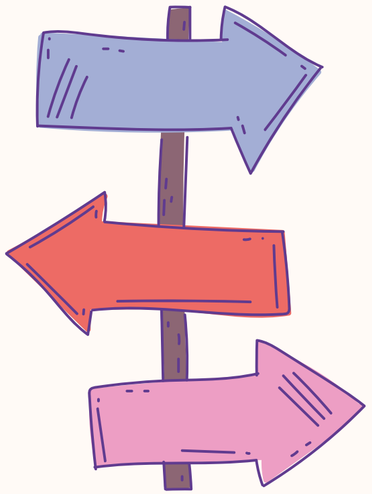
Backend Architecture

```
backend/
├── controllers/              # Business logic layer
│   ├── auth_controller.py    # Authentication operations
│   ├── community_controller.py  # Community club operations
│   ├── donations_controller.py  # Donation management
│   ├── inventory_controller.py  # Inventory tracking and management
│   ├── jobs_controller.py    # Background job scheduling
│   ├── notification_controller.py # Notification management
│   ├── profile_controller.py    # User profile management
│   └── requests_controller.py   # Request handling
├── services/                 # Domain logic layer
│   ├── auth_strategies.py    # Strategy pattern for authentication
│   ├── community_clubs.py    # Community club management
│   ├── find_user.py          # User lookup utilities
│   ├── image_upload.py       # Image upload handling
│   ├── jobs_service.py       # Background job scheduling
│   ├── metrics.py            # Analytics and metrics
│   ├── notification_service.py  # Notification management
│   ├── notification_strategies.py # Observer pattern for notifications
│   ├── password.py           # Secure password hashing
│   └── run_allocation.py     # Smart matching algorithm
├── database/              # Data access layer
│   ├── database_factory.py   # Factory pattern for DB selection
│   ├── database_interface.py    # Abstract database interface
│   ├── postgres_database.py  # PostgreSQL implementation
│   └── sqlite_database.py    # SQLite implementation
├── routes/                # API endpoint layer
│   ├── auth_routes.py        # Authentication endpoints
│   ├── community_routes.py   # Community club endpoints
│   ├── donations_routes.py   # Donation API routes
│   ├── inventory_routes.py   # Inventory management endpoints
│   ├── jobs_routes.py        # Background job endpoints
│   ├── notification_routes.py   # Notification endpoints
│   ├── profile_routes.py     # User profile endpoints
│   └── requests_routes.py    # Request API routes
├── models.py              # SQLAlchemy data models
├── config.py              # Application configuration
├── extensions.py          # Flask extensions setup
├── broadcast_observer.py # Observer pattern implementation
└── app.py                 # Application factory
```

Frontend Architecture

```
frontend/
├── src/
│   ├── assets/           # Static assets
│   │   └── logo.png              # Application logo
│   ├── Components/       # Reusable React components
│   │   ├── CommunityClubsMap.jsx # Interactive map component
│   │   ├── DonationCard.jsx      # Donation display card
│   │   └── TopNav.jsx            # Top navigation bar
│   ├── pages/            # Page-level components
│   │   ├── ClientUI/             # Client-specific pages
│   │   │   ├── ClientDashboard.jsx # Client analytics dashboard
│   │   │   ├── ClientHome.jsx    # Client main page
│   │   │   ├── DonationForm.jsx  # Create donation form
│   │   │   ├── DonationsList.jsx # View donations list
│   │   │   ├── RegisterForm.jsx  # User registration form
│   │   │   ├── RequestForm.jsx   # Create request form
│   │   │   └── RequestsList.jsx  # View requests list
│   │   ├── MainUI/               # Shared UI components
│   │   │   ├── Login.jsx         # Authentication page
│   │   │   ├── Notifications.jsx # Notifications center
│   │   │   ├── Profile.jsx       # User profile management
│   │   │   └── Subscriptions.jsx # Community subscriptions
│   │   └── ManagerUI/            # Manager-specific pages
│   │       ├── CompleteRequest.jsx # Complete request workflow
│   │       ├── ManageDonations.jsx # Donation approval system
│   │       ├── ManagerAction.jsx   # Manager action center
│   │       ├── ManagerDashboard.jsx # Manager analytics dashboard
│   │       ├── ManageRegistrations.jsx # User registration approval
│   │       └── ManagerHome.jsx      # Manager main page
│   ├── styles/           # CSS styling
│   │   ├── ClientDashboard.css   # Client dashboard styles
│   │   ├── ClientHome.css        # Client home styles
│   │   ├── CommunityClubsMap.css # Map component styles
│   │   ├── CompleteRequest.css   # Request completion styles
│   │   ├── DonationsList.css     # Donations list styles
│   │   ├── Login.css             # Login page styles
│   │   ├── ManageDonations.css   # Donation management styles
│   │   ├── ManagerAction.css     # Manager action styles
│   │   ├── ManagerDashboard.css  # Manager dashboard styles
│   │   ├── ManageRegistrations.css # Registration management styles
│   │   ├── ManagerHome.css       # Manager home styles
│   │   ├── Notifications.css     # Notifications styles
│   │   ├── Profile.css           # Profile page styles
│   │   ├── RegisterForm.css      # Registration form styles
│   │   ├── RequestsList.css      # Requests list styles
│   │   └── TopNav.css            # Navigation bar styles
│   ├── App.jsx           # Main application component
│   ├── httpClient.jsx    # Axios HTTP client configuration
│   └── main.jsx          # Application entry point
├── package.json          # Dependencies and scripts
├── vite.config.js        # Vite build configuration
└── index.html            # HTML template
```

FILE STRUCTURES



DOCUMENTATION

```python
def run_cleanup_approved_donations_once(days_until_delete: int = 2) -> Dict[str, str]:
    """Clean up old approved donations that haven't been collected.

    Delete donations with status == 'Approved' that are older than N days.
    Also, send a notification to the donor about removal.

    Args:
        days_until_delete (int): Number of days before deleting approved donations.

    Returns:
        dict: Job execution results with deletion count.
    """
    now_utc = datetime.now(timezone.utc)
    cutoff = now_utc - timedelta(days=days_until_delete)

    # Find old approved donations
    old_donations: List[Donation] = Donation.query.filter(
        Donation.status == "Approved",
        Donation.approved_at <= cutoff
    ).all()

    if not old_donations:
        return {"job": "cleanup_approved_donations", "status": "ok", "deleted": 0, "at": now_utc.isoformat()}

    deleted_count = 0
    for donation in old_donations:
        # Create notification for the donor
        message = (
            f"Your donation '{donation.donation_item}' in {donation.location} "
            "has been automatically removed after 2 days of approval."
        )
        notification_strategy = DatabaseNotificationStrategy()
        notification_strategy.create_notification(message=message, receiver_email=donation.donor_email)

        # Delete donation (cascade removes items, reservations)
        db.session.delete(donation)
        deleted_count += 1

    db.session.commit()

    return {
        "job": "cleanup_approved_donations",
        "status": "ok",
        "deleted": deleted_count,
        "at": now_utc.isoformat(),
    }
```

```python
def run_allocation() -> Dict[str, str]:
    """Match pending requests to available items using FIFO algorithm.

    Matches Pending requests to Available items (FIFO within each queue).
    Updates request status to 'Matched' when fully allocated and sends
    notifications to requesters.

    Returns:
        dict: Allocation job execution results.
    """
    # Get current date in Singapore timezone for expiry checks
    sg_today = datetime.now(SG_TZ).date()
    now_utc = datetime.now(timezone.utc)

    # Get all pending requests ordered by creation time (FIFO)
    pending: List[Request] = (Request.query
        .filter(Request.status == "Pending")
        .order_by(Request.created_at.asc(), Request.id.asc())
        .all())

    changed = False  # Track if any changes were made

    # Process each pending request in FIFO order
    for req in pending:
        requested = (req.request_quantity or 0)
        allocated = (req.allocation or 0)
        need = max(0, requested - allocated)  # How many more items needed

        # If request is already fully allocated, mark as Matched
        if need == 0:
            if requested > 0 and req.status != "Matched":
                req.status = "Matched"
                req.matched_at = now_utc
                changed = True
            continue  # Move to next request

        # Find matching available items for this request
        # Match criteria: same location, same item type, not expired
        candidates = (db.session.query(Item)
            .join(Donation, Item.donation_id == Donation.id)
            .filter(
                Item.status == "Available",                    # Item must be available
                Donation.location == req.location,             # Same community club
                Donation.donation_item == req.request_item,    # Same item type
                # Either no expiry date or not yet expired in Singapore timezone
                or_(Donation.expiryDate.is_(None), Donation.expiryDate >= sg_today),
            )
            .order_by(Item.id.asc())  # FIFO allocation (earliest items first)
            .limit(need)              # Only get as many as needed
            .all())

        # Allocate each candidate item to this request
        for it in candidates:
            it.status = "Unavailable"  # Mark item as allocated
            db.session.add(Reservation(request_id=req.id, item_id=it.id))  # Create reservation
            req.allocation = (req.allocation or 0) + 1  # Increment allocation count
            changed = True

        # If request is now fully allocated, mark as Matched and notify user
        if (req.allocation or 0) >= requested and requested > 0:
            req.status = "Matched"
            req.matched_at = now_utc
            changed = True

            # Send notification to requester about successful match
            message = (
                f"Good news! Your request '{req.request_item}' in {req.location} "
                "has been successfully matched with available items."
            )
            notification_strategy = DatabaseNotificationStrategy()
            notification_strategy.create_notification(message=message, receiver_email=req.requester_email)

    # Commit all changes if any allocations were made
    if changed:
        db.session.commit()

    return {"job": "allocation", "status": "ok", "at": now_utc.isoformat()}
```

Improves code-readability

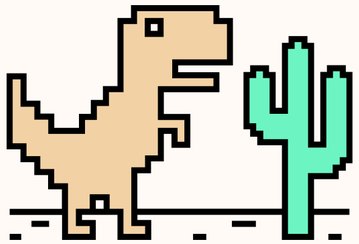
CODE COMMENTS

SECURITY

```python
def init_session(app, session_type, redis_url, env):
    """Initialize Flask session configuration.

    Args:
        app (Flask): Flask application instance.
        session_type (str): Session storage type ('redis' or 'filesystem').
        redis_url (str): Redis connection URL for redis session type.
        env (str): Environment ('production' or 'development').
    """
    from redis import from_url as redis_from_url
    app.config["SESSION_TYPE"] = session_type

    # Configure Redis for session storage if specified
    if session_type == "redis":
        app.config["SESSION_REDIS"] = redis_from_url(redis_url)

    # Set session cookie configuration
    app.config["SESSION_COOKIE_NAME"] = "session"

    # Configure cookie security based on environment
    if env == "production":
        app.config["SESSION_COOKIE_SAMESITE"] = "None"  # Allow cross-site cookies
        app.config["SESSION_COOKIE_SECURE"] = True      # Require HTTPS
    else:
        app.config["SESSION_COOKIE_SAMESITE"] = "Lax"   # Same-site for development
        app.config["SESSION_COOKIE_SECURE"] = False     # Allow HTTP in development

    app.config["SESSION_PERMANENT"] = False  # Sessions expire when browser closes
    FlaskSession(app)
```

```python
def get_current_user():
    """Get the currently authenticated user from session.

    Returns:
        User: Current user instance or None if not authenticated.
    """
    email = session.get("user_email")
    if not email:
        return None
    return db.session.get(User, email)
```

```javascript
// authenticate
useEffect(() => {
  (async () => {
    try {
      const resp = await httpClient.get("/api/@me");
    } catch (error) {
      alert("Not authenticated");
      navigate("/login");
    }
  })();
}, []);
```



USER SESSION MANAGEMENT

Redis-backed sessions ensure safe access

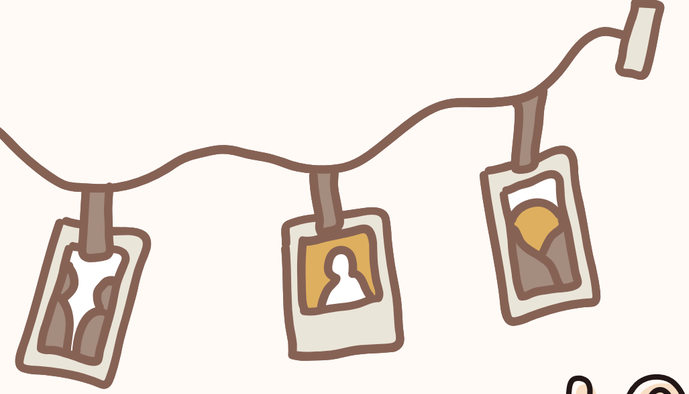
SECURITY


```python
from passlib.hash import argon2

def hash_password(pw: str) -> str:
    """Hash a password using Argon2.

    Args:
        pw (str): Plain text password to hash.

    Returns:
        str: Hashed password string.
    """
    return argon2.hash(pw)
```

Argon2 offers top-tier password security through high memory usage, tunable cost, and resistance to modern cracking techniques

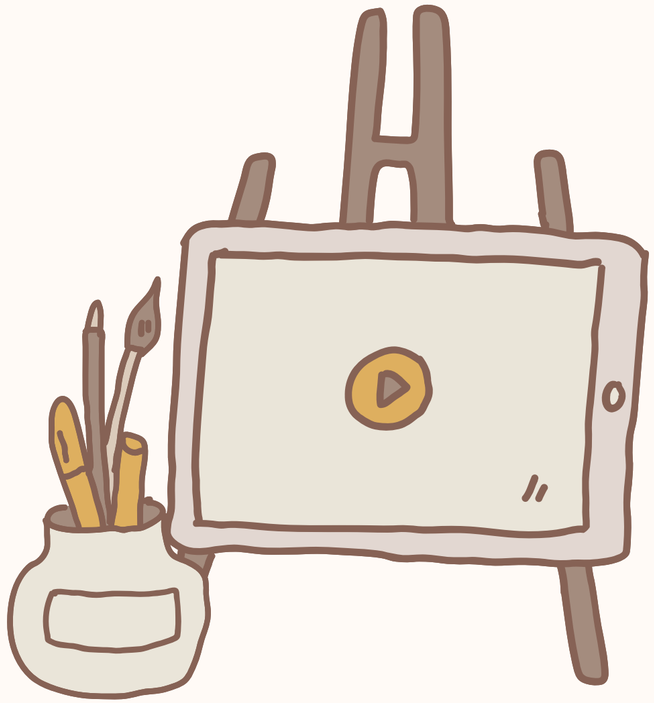
PASSWORD HASHING

TRACEABLITY IN PROJECT DELIVERABLES



LOGIN

USE CASE DESCRIPTION

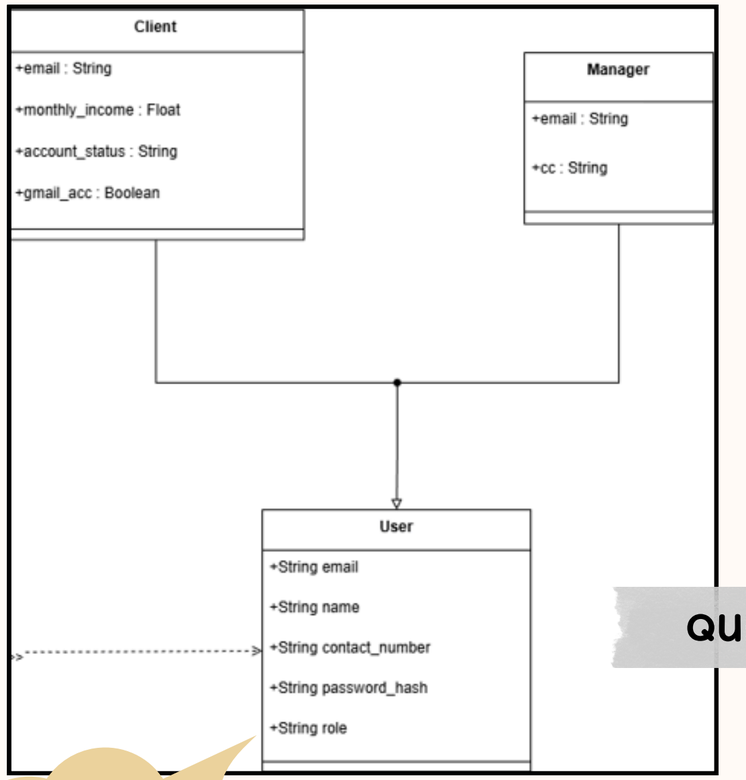


UC 1.1.1 - Login

Use Case ID:	UC 1.1.1		
Use Case Name:	Login		
Created By:	Aaron	Last Updated By:	
Date Created:	03.11.25	Date Last Updated:	

Actor:	User
Description:	Standard authentication flow (email/password or third-party); grants access tokens and loads user profile and role.
Preconditions:	1. User has an account.
Postconditions:	1. Session established; role determines UI features.
Priority:	High
Frequency of Use:	Several times per week.
Flow of Events:	1. User enters credentials 2. System authenticates: 3. If success: a. Redirect to appropriate landing.
Alternative Flows:	1.2.1.AC.1: If authentication, unsuccessful, try again
Exceptions:	-
Includes:	-
Special Requirements:	1. Brute-force protections 2. Session timeout.
Assumptions:	-
Notes and Issues:	-

LOGIN
CLASS DIAGRAMS



User **class holds sensitive information like** email **and** password_hash**, used for authentication checks**

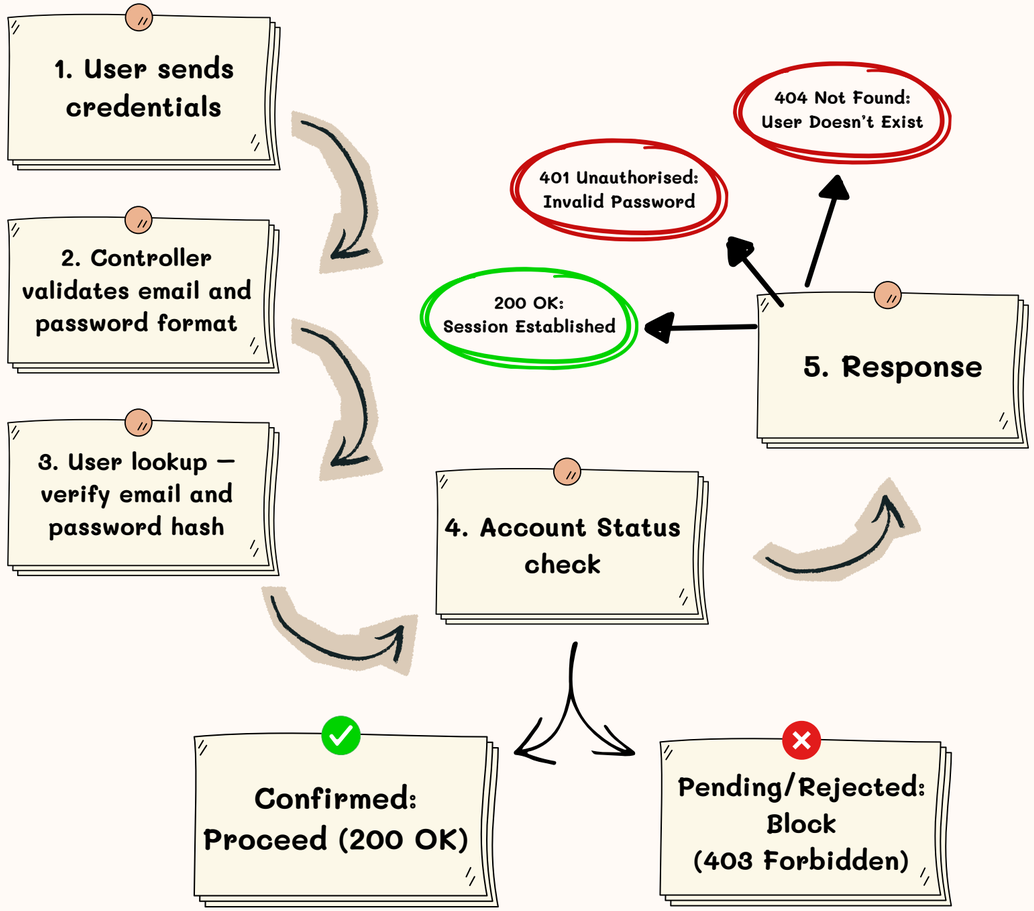
QUERIES USER BY EMAIL

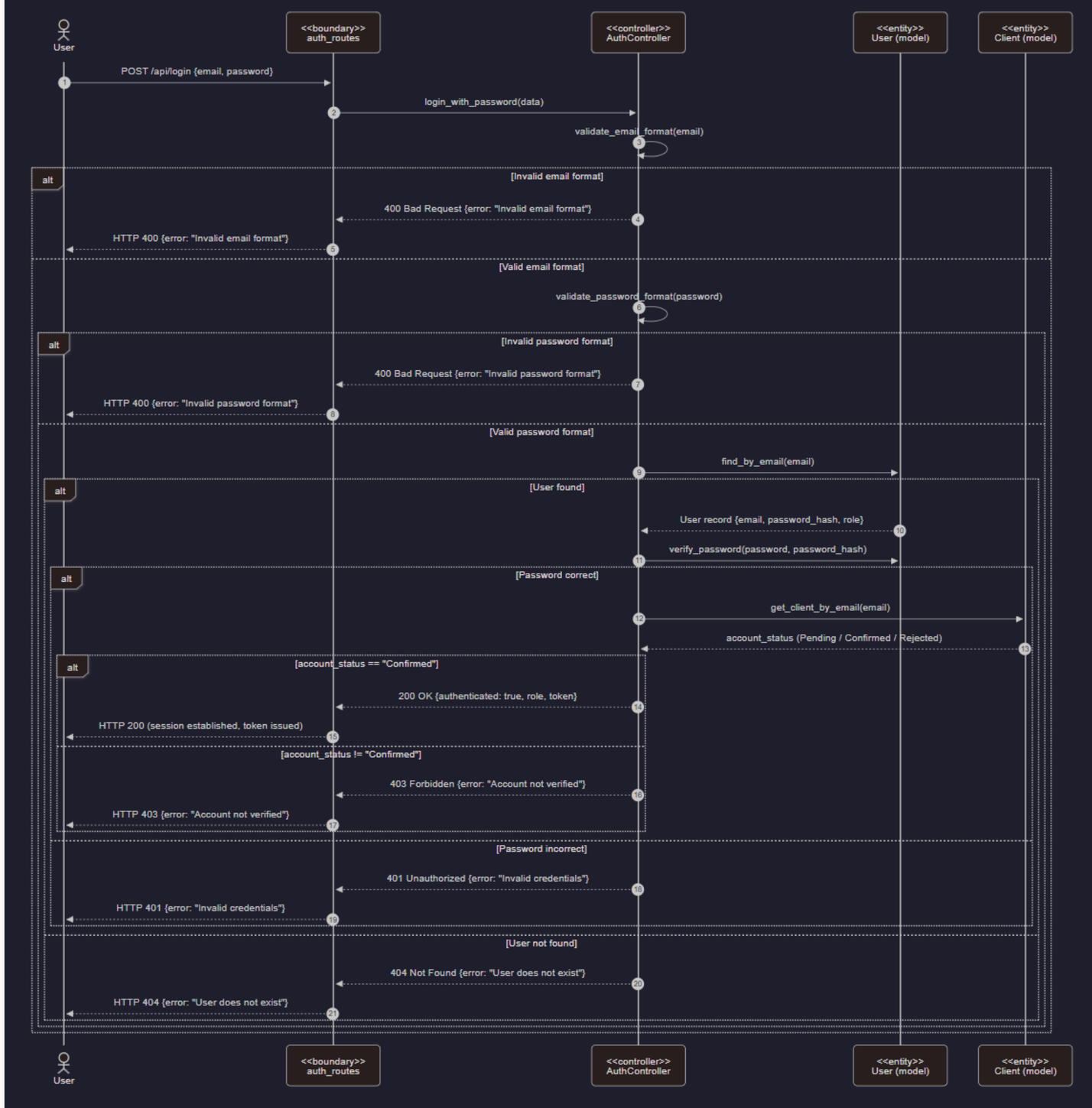
AuthController
+start_google_login()
+google_callback()
+login_with_password()
+register_user()

- AuthController **queries the** User **class by email to authenticate**
- **Validates the password against the hash value stored in the** User **model**
- **Determines access based on the role and account status**

LOGIN

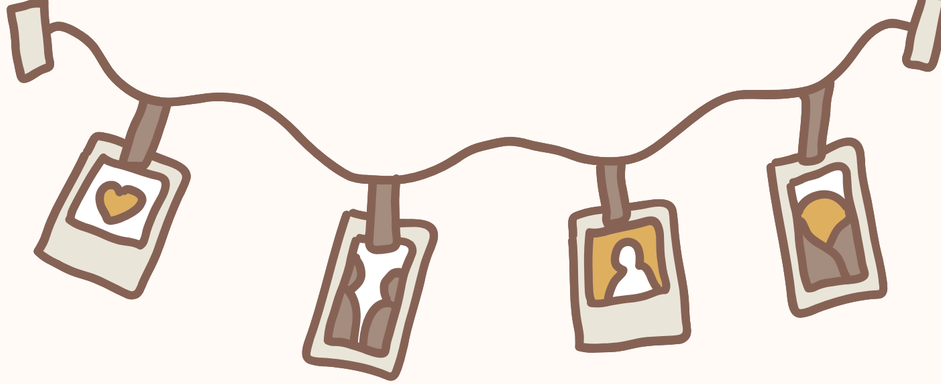
SEQUENCE DIAGRAMS





LOGIN

GOOD DESIGN APPLICATION - FACADE PATTERN



auth_routes.py

```python
# Google OAuth login initiation
@auth_bp.get("/login")  wenhong
def login_google():
    return c.start_google_login()

# Google OAuth callback handler
@auth_bp.get("/auth/callback")  wenhong
def auth_callback():
    return c.google_callback(Config.FRONTEND_ORIGIN)

# User registration with email/password
@auth_bp.post("/register")  wenhong
def register():
    return c.register_user(request.get_json(force=True, silent=True) or {})

# Email/password login
@auth_bp.post("/login_password")  wenhong
def login_password():
    return c.login_with_password(request.get_json(force=True, silent=True) or {})

# User logout (clears session)
@auth_bp.post("/logout")  wenhong
def logout():
    return c.logout_user()
```

Hides the complexity and provides a simplified interface for the client to interact with

AuthController

```python
class AuthController:  1 usage  wenhong
    """Controller class for handling authentication operations.

    Uses the Strategy pattern to support multiple authentication methods
    including Google OAuth and password-based authentication.
    """
    def __init__(self):  wenhong
        self.auth_context = AuthenticationContext()
```

login_with_password()

```python
@staticmethod  1 usage  wenhong
def login_with_password(data):
    """Login using password strategy.

    Args:
        data (dict): Login credentials including email and password.

    Returns:
        tuple: JSON response and HTTP status code.
    """
    auth_context = AuthenticationContext()
    auth_context.set_strategy(PasswordStrategy())

    result, status_code = auth_context.authenticate(data)
    return jsonify(result), status_code
```

LOGIN TESTS

Cyclomatic Complexity

- A software metric used to measure the complexity of a program
- C.C. = | Decision Points | + 1
  = 3 + 1 = 4

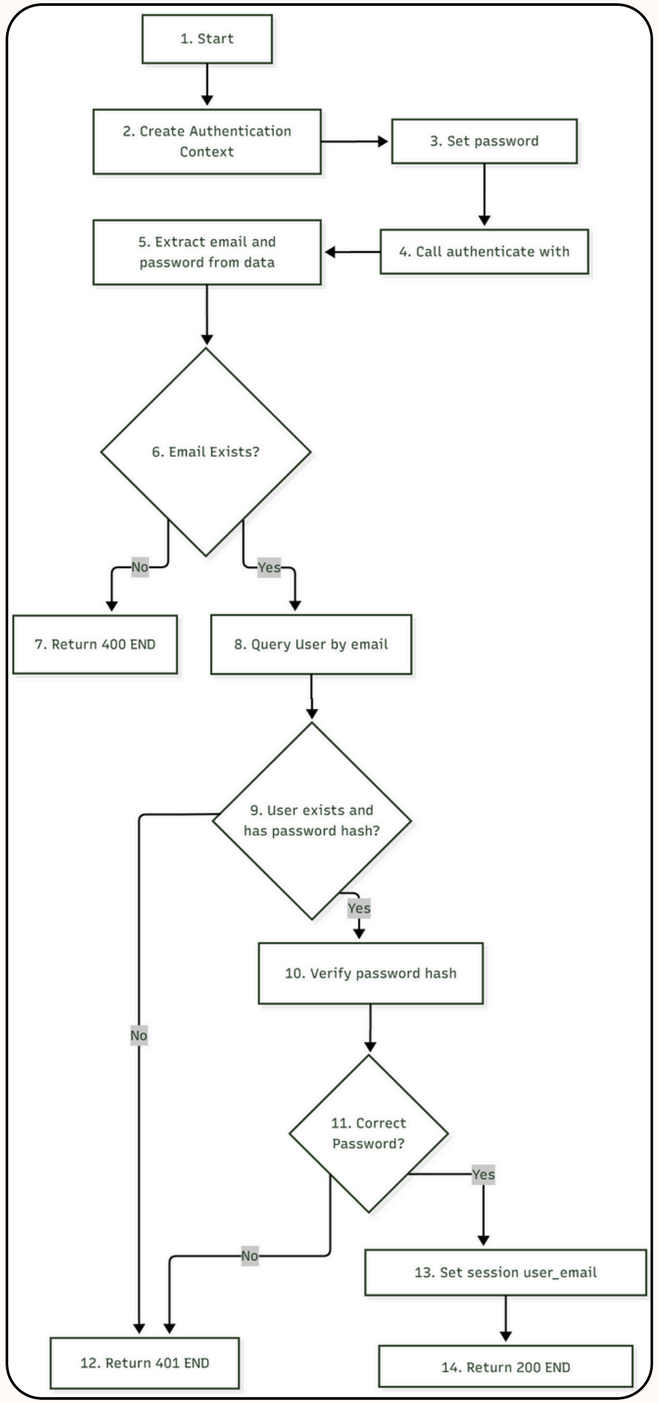
Basis Paths

Path 1: 1 - 2 - 3 - 4 - 5 - 6 - 7

Path 2: 1 - 2 - 3 - 4 - 5 - 6 - 8 - 9 - 12

Path 3: 1 - 2 - 3 - 4 - 5 - 6 - 8 - 9 - 10 - 11 - 12

Path 4: 1 - 2 - 3 - 4 - 5 - 6 - 8 - 9 - 10 - 11 - 13 - 14



LOGIN
TEST CASES

TEST CASES				
NO.	TEST INPUT	EXPECTED OUTPUT	ACTUAL OUTPUT	PASS?
1	{"EMAIL": "USER@EXAMPLE.COM", "PASSWORD": "VALIDPASS123}	{"AUTHENTICATED": TRUE, "ROLE": "C"}, 200	{"AUTHENTICATED": TRUE, "ROLE": "C"}, 200	PASS ✅
2	{"EMAIL": "INVALID.EMAIL", "PASSWORD": "VALIDPASS123"}	{"ERROR": "EMAIL AND PASSWORD REQUIRED"}, 400	{"ERROR": "EMAIL AND PASSWORD REQUIRED"}, 400	PASS ✅
3	{"EMAIL": "USER@TEST.COM", "PASSWORD": "SHORT"	{"ERROR": "EMAIL AND PASSWORD REQUIRED"}, 400	{"ERROR": "EMAIL AND PASSWORD REQUIRED"}, 400	PASS ✅
4	{"EMAIL": "NONEXISTENT@TEST.COM", "PASSWORD": "VALIDPASS123"}	{"ERROR": "INVALID CREDENTIALS"}, 401	{"ERROR": "INVALID CREDENTIALS"}, 401	PASS ✅
5	{"EMAIL": "USER@TEST.COM", "PASSWORD": "WRONGPASS123"}	{"ERROR": "INVALID CREDENTIALS"}, 401	{"ERROR": "INVALID CREDENTIALS"}, 401	PASS ✅
6	{"EMAIL": "", "PASSWORD": ""}	{"ERROR": "EMAIL AND PASSWORD REQUIRED"}, 400	{"ERROR": "EMAIL AND PASSWORD REQUIRED"}, 400	PASS ✅

FUTURE PLANS

AI-POWERED IMAGE VALIDATOR

CHATBOT ASSISTANCE

SOCIAL FEATURES


THANK
YOU