Software Design Document

Face Recognition Attendance System

Built with FastAPI (Python) & React (TypeScript)

Date: December 2025

Junior Software Engineering Project

Contents

1 Introduction

1.1 Purpose

This Software Design Document (SDD) provides a comprehensive architectural overview of the **Face Recognition Attendance System**. The system automates attendance tracking in educational institutions using AI-powered facial recognition technology, eliminating manual roll calls and reducing administrative overhead.

1.2 Scope

The system encompasses:

- Real-time face recognition for attendance marking
- Role-based access control (Admin, Mentor, Student)
- Course and class schedule management
- Attendance session management with state machine
- Real-time notifications via Observer pattern
- Comprehensive reporting and analytics

1.3 Technology Stack

Component	Technology
Backend Framework	FastAPI (Python 3.11+)
Frontend Framework	React 18 with TypeScript
Database	PostgreSQL with SQLAlchemy ORM
AI/ML	dlib, face_recognition library
Authentication	JWT (JSON Web Tokens)
UI Components	shadcn/ui, Tailwind CSS
State Management	React Context API

Table 1: Technology Stack Overview

1.4 Project Team

The following team members are responsible for the design and implementation of this project:

Student ID	Student Name
2023/08770	Seif El Din Ashraf
2023/02630	Omar Abdelghany Mahmoud
2023/07019	Abdulkader Adnan Ibrahim
2023/03897	Ammar Yasser Mohamed
2023/06891	Adham Hossam El Din

Table 2: Project Team Members

2 Design Viewpoints

2.1 System Architecture Diagram

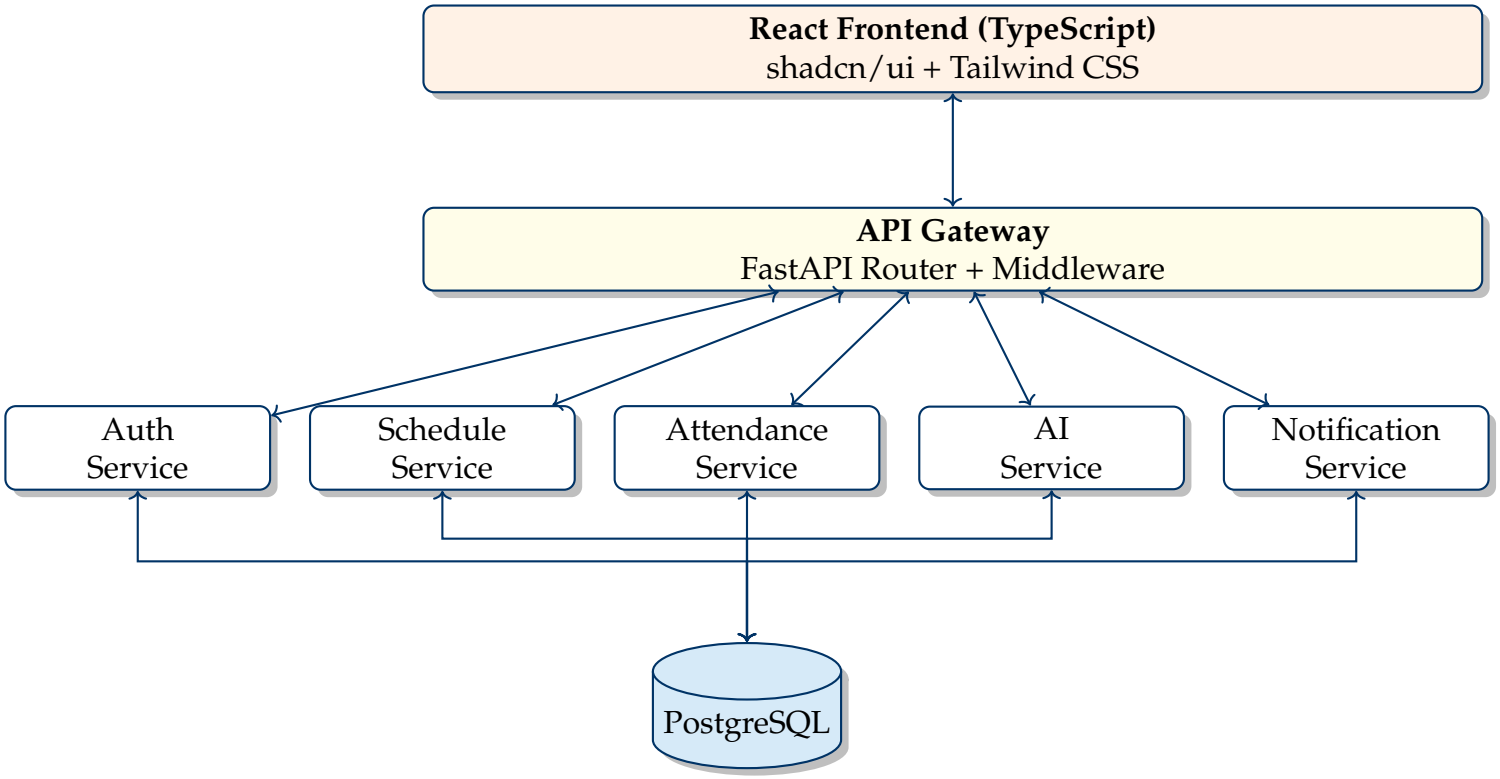

Figure 1: Microservices Architecture Overview

2.2 Class Diagram - Core Entities

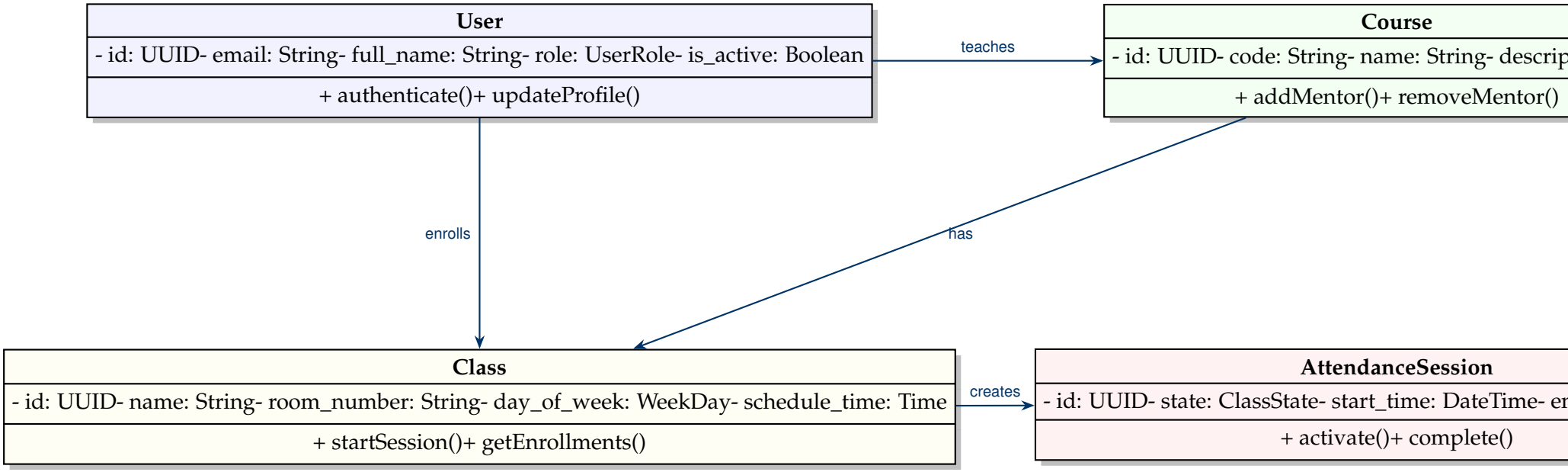

Figure 2: Core Domain Class Diagram

2.3 Use Case Diagram

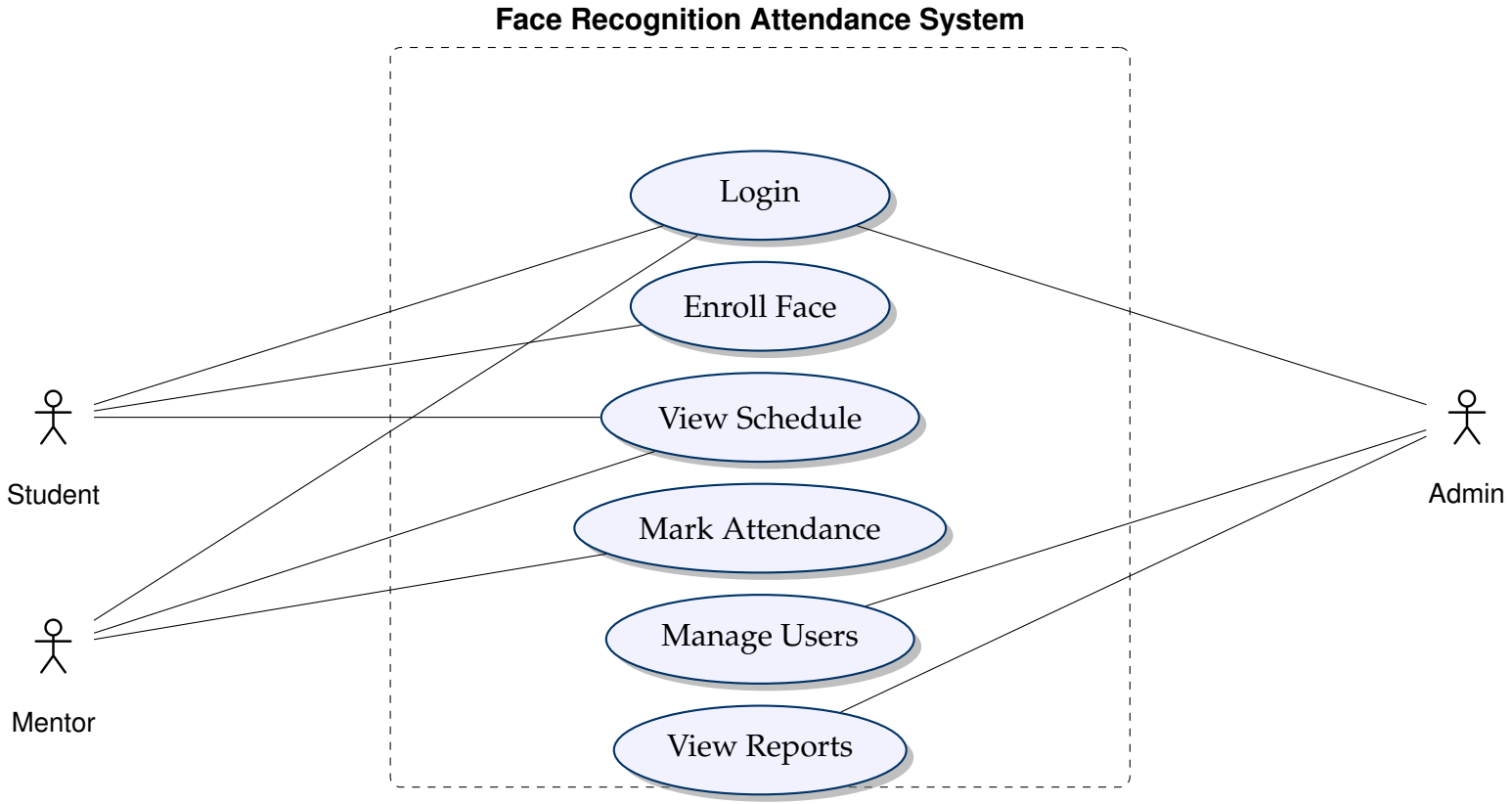

Figure 3: Use Case Diagram

2.4 Sequence Diagram - Face Recognition Attendance

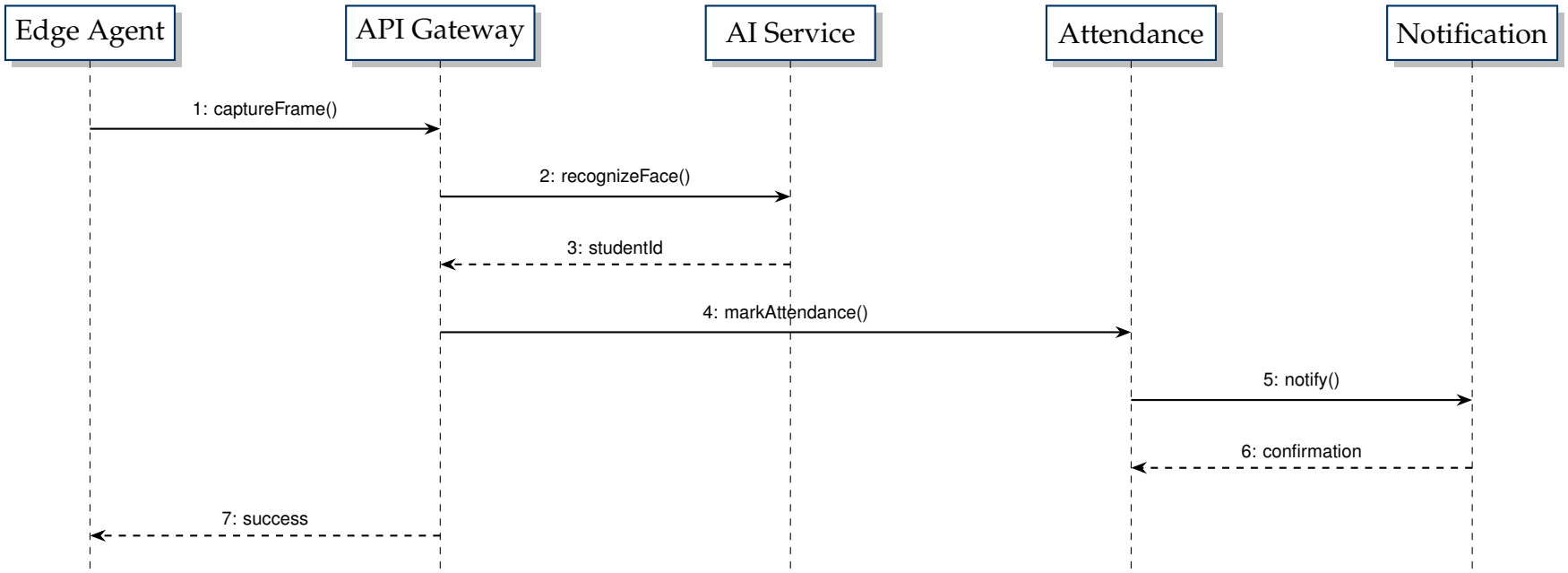

Figure 4: Face Recognition Attendance Sequence

3 Data Design

3.1 Database Schema Overview

The system uses PostgreSQL with a normalized relational schema. The database is organized into logical domains corresponding to each microservice.

3.2 Entity Relationship Description

Table	Service	Description
users	Auth	Stores all system users (students, mentors, admins)
api_keys	Auth	API key management for external integrations
courses	Schedule	Course catalog with code and description
course_mentors	Schedule	Many-to-many relationship between courses and mentors
classes	Schedule	Scheduled class sessions with room and time
enrollments	Schedule	Student enrollment in classes
face_encodings	AI	Facial feature vectors for recognition
attendance_sessions	Attendance	Active/completed attendance sessions
attendance_records	Attendance	Individual attendance marks per student
notifications	Notification	User notifications with read status

Table 3: Database Tables Overview

3.3 Key Relationships

- **User** $\leftrightarrow$ **Course**: Many-to-many via `course_mentors` (mentors can teach multiple courses)
- **User** $\leftrightarrow$ **Class**: Many-to-many via `enrollments` (students enroll in classes)
- **Course** $\rightarrow$ **Class**: One-to-many (a course has multiple class sections)
- **Class** $\rightarrow$ **AttendanceSession**: One-to-many (each class can have multiple sessions)
- **User** $\rightarrow$ **FaceEncoding**: One-to-many (multiple face samples per user)

3.4 Data Types and Enums

```
CREATE TYPE user_role AS ENUM ('student', 'mentor', 'admin');
CREATE TYPE class_state AS ENUM ('inactive', 'active', 'completed');
CREATE TYPE attendance_status AS ENUM ('present', 'absent', 'late', 'excused');
CREATE TYPE notification_type AS ENUM ('class_started', 'attendance_marked', 'announcement');
CREATE TYPE week_day AS ENUM ('monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'saturday', 'sunday
    ');
```

Listing 1: Custom PostgreSQL Enums

4 User Interface Design

4.1 Design Philosophy

The UI follows modern design principles:

- **Component-Based**: Built with shadcn/ui for consistent, accessible components
- **Responsive**: Mobile-first design using Tailwind CSS
- **Role-Based Views**: Different dashboards for Admin, Mentor, and Student
- **Dark/Light Theme**: Support for user preference
- **Real-time Updates**: Live notifications and attendance status

4.2 Key UI Components

Component	Purpose
Dashboard	Role-specific overview with stats and quick actions
Sidebar	Navigation with role-based menu items
Schedule Table	Weekly view of classes with day columns
User Management	CRUD interface with confirmation dialogs
Face Enrollment	Camera capture with real-time preview
Attendance Session	Live face recognition with student list

Table 4: Key UI Components

4.3 Navigation Structure

- **Admin**: Dashboard, Users, Courses, Classes, Attendance, Schedule, Notifications
- **Mentor**: Dashboard, Courses, Attendance, Schedule, Notifications
- **Student**: Dashboard, Face Enrollment, Attendance, Schedule, Notifications

5 Project Plan

5.1 Team Roles

Role	Responsibility	Deliverables
Backend Developer	API development, database design	FastAPI services, SQLAlchemy models
Frontend Developer	UI implementation, state management	React components, API integration
AI/ML Engineer	Face recognition implementation	dlib integration, encoding pipeline
DevOps Engineer	Deployment, CI/CD	Docker configs, deployment scripts
QA Engineer	Testing, documentation	Test cases, SDD document

Table 5: Team Roles and Responsibilities

5.2 Development Timeline

Phase	Duration	Milestones
Phase 1: Foundation	Weeks 1-2	Database schema, Auth service, basic UI
Phase 2: Core Features	Weeks 3-5	Schedule service, Attendance service
Phase 3: AI Integration	Weeks 6-7	Face recognition, Edge agent
Phase 4: Polish	Weeks 8-9	Notifications, Reports, UI refinement
Phase 5: Testing	Week 10	Integration testing, Documentation

Table 6: Development Timeline

6 Implementation Details

6.1 Unit Testing

The project employs comprehensive testing strategies:

```python
import pytest
from services.auth_service.services.auth_service import AuthService

class TestAuthService:
    def test_password_hashing(self):
        service = AuthService(db_session)
        hashed = service.hash_password("Password123")
        assert service.verify_password("Password123", hashed)

    def test_jwt_token_generation(self):
        token = service.create_access_token(user_id="123")
        decoded = service.decode_token(token)
        assert decoded["sub"] == "123"
```

Listing 2: Example Unit Test

6.2 MVC Architecture

The system follows a modified MVC pattern adapted for microservices:

- **Model**: SQLAlchemy ORM models in `/models` directories
- **View**: React components in `/frontend/src/pages`
- **Controller**: FastAPI route handlers in `/api/routes.py`

Each service follows a layered architecture:

```
service/
 api/            # Route handlers (Controller)
 models/         # SQLAlchemy models (Model)
 repositories/   # Data access layer
 schemas/        # Pydantic validation schemas
 services/       # Business logic layer
```

6.3 Data Validation

Input validation is handled at multiple levels:

```python
from pydantic import BaseModel, EmailStr, validator
class UserCreate(BaseModel):
    email: EmailStr
    password: str
    full_name: str
    role: UserRole = UserRole.STUDENT

    @validator('password')
    def password_strength(cls, v):
        if len(v) < 8:
            raise ValueError('Password must be at least 8 characters')
        if not any(c.isupper() for c in v):
            raise ValueError('Password must contain uppercase')
        return v
```

Listing 3: Pydantic Schema Validation

Frontend validation mirrors backend rules:

```javascript
const validatePassword = (password: string): string | null => {
    if (password.length < 8)
        return 'Password must be at least 8 characters';
    if (!/[A-Z]/.test(password))
        return 'Must contain uppercase letter';
    if (!/[a-z]/.test(password))
        return 'Must contain lowercase letter';
    if (!/\d/.test(password))
        return 'Must contain a digit';
    return null;
};
```

Listing 4: Frontend Validation

6.4 Clean Code Principles

The codebase adheres to clean code standards:

- **Single Responsibility**: Each service handles one domain
- **Dependency Injection**: Services receive dependencies via constructors
- **Type Hints**: Full Python type annotations and TypeScript types
- **Meaningful Names**: Descriptive function and variable names
- **Small Functions**: Functions do one thing well
- **Error Handling**: Consistent exception handling with custom exceptions

6.5 CRUD Operations

Standard CRUD operations are implemented via repository pattern:

```
class UserRepository:
    def _init_(self, db: Session):
        self.db = db

    def create(self, user_data: UserCreate) -> User:
        user = User(**user_data.dict())
        self.db.add(user)
        self.db.commit()
        return user
```

```
    def find_by_id(self, user_id: UUID) -> Optional[User]:
        return self.db.query(User).filter(User.id == user_id).first()

    def update(self, user_id: UUID, data: dict) -> User:
        user = self.find_by_id(user_id)
        for key, value in data.items():
            setattr(user, key, value)
        self.db.commit()
        return user

    def delete(self, user_id: UUID) -> bool:
        user = self.find_by_id(user_id)
        self.db.delete(user)
        self.db.commit()
        return True
```

Listing 5: Repository Pattern Example

7 Object-Oriented Programming & Design Patterns

7.1 OOP Principles Applied

7.1.1 Encapsulation

Each service encapsulates its data and behavior:

```
class AttendanceService:
    def _init_(self, db: Session):
        self._session_repo = SessionRepository(db)
        self._record_repo = RecordRepository(db)

    def mark_attendance(self, session_id: UUID, student_id: UUID):
        # Internal logic hidden from callers
        session = self._session_repo.find_by_id(session_id)
        if session.state != ClassState.ACTIVE:
            raise InvalidStateError("Session not active")
        return self._record_repo.create(session_id, student_id)
```

7.1.2 Inheritance

Base repository class provides common functionality:

```
class BaseRepository(Generic[T]):
    model: Type[T]

    def find_all(self, skip: int = 0, limit: int = 100) -> List[T]:
        return self.db.query(self.model).offset(skip).limit(limit).all()

class UserRepository(BaseRepository[User]):
    model = User
```

7.1.3 Polymorphism

Authentication strategies demonstrate polymorphism:

```python
class AuthStrategy(ABC):
    @abstractmethod
    def authenticate(self, credentials: dict) -> User:
        pass

class PasswordAuthStrategy(AuthStrategy):
    def authenticate(self, credentials: dict) -> User:
        # Password-based authentication
        pass

class APIKeyAuthStrategy(AuthStrategy):
    def authenticate(self, credentials: dict) -> User:
        # API key authentication
        pass
```

7.2 Design Pattern 1: State Pattern

The attendance session uses the State Pattern to manage session lifecycle:

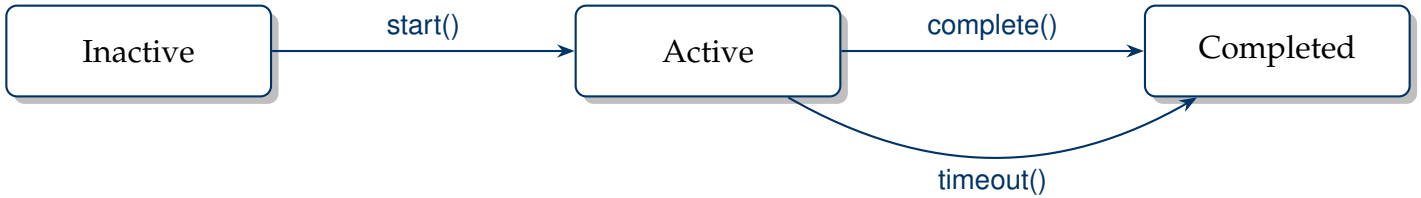

Figure 5: Attendance Session State Machine

```python
class SessionState(ABC):
    @abstractmethod
    def start(self, session: AttendanceSession) -> None:
        pass

    @abstractmethod
    def mark_attendance(self, session: AttendanceSession, student_id: UUID) -> None:
        pass

class InactiveState(SessionState):
    def start(self, session: AttendanceSession) -> None:
        session.state = ClassState.ACTIVE
        session.start_time = datetime.now()

    def mark_attendance(self, session: AttendanceSession, student_id: UUID) -> None:
        raise InvalidStateError("Cannot mark attendance on inactive session")

class ActiveState(SessionState):
    def start(self, session: AttendanceSession) -> None:
        raise InvalidStateError("Session already active")

    def mark_attendance(self, session: AttendanceSession, student_id: UUID) -> None:
        # Create attendance record
        pass
```

Listing 6: State Pattern Implementation

7.3 Design Pattern 2: Observer Pattern

The notification system implements the Observer Pattern for real-time updates:

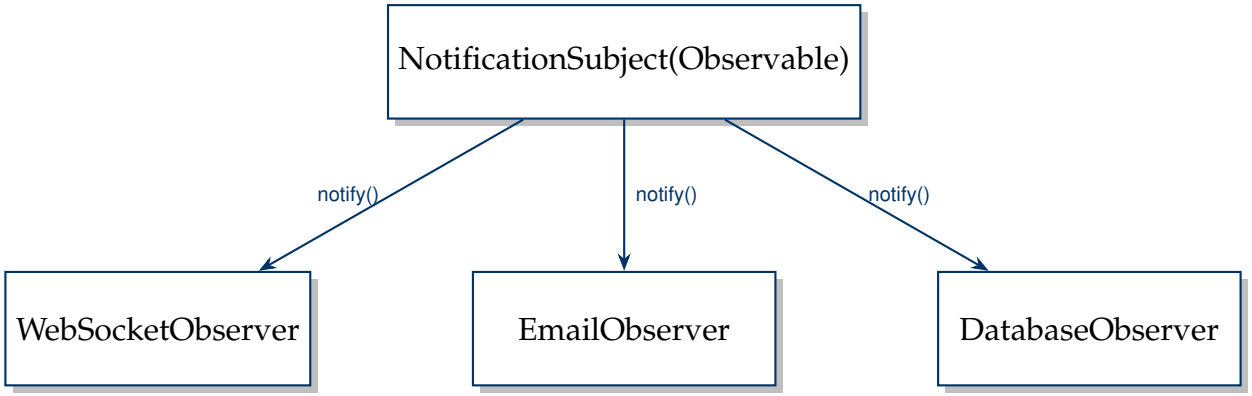

Figure 6: Observer Pattern for Notifications

```python
class NotificationObserver(ABC):
    @abstractmethod
    def update(self, notification: Notification) -> None:
        pass

class NotificationSubject:
    def _init_(self):
        self._observers: List[NotificationObserver] = []

    def attach(self, observer: NotificationObserver) -> None:
        self._observers.append(observer)

    def notify(self, notification: Notification) -> None:
        for observer in self._observers:
            observer.update(notification)

class DatabaseObserver(NotificationObserver):
    def update(self, notification: Notification) -> None:
        # Persist notification to database
```

```
        self.repo.create(notification)

class WebSocketObserver(NotificationObserver):
    def update(self, notification: Notification) -> None:
        # Push to connected WebSocket clients
        self.ws_manager.broadcast(notification.user_id, notification)
```

Listing 7: Observer Pattern Implementation

7.4 Additional Patterns Used

Pattern	Location	Purpose
Repository	All services	Abstract data access layer
Factory	Notification service	Create different notification types
Strategy	Auth service	Multiple authentication methods
Singleton	Database connection	Single database connection pool
Adapter	AI service	Wrap face_recognition library

Table 7: Design Patterns Summary

8 Advanced Features

8.1 Dynamic Menu System

The sidebar navigation dynamically adjusts based on user role:

```
const menuItems = [
  { name: 'Dashboard', icon: Home, href: '/dashboard', roles: ['admin', 'mentor', 'student'] },
  { name: 'Users', icon: Users, href: '/users', roles: ['admin'] },
  { name: 'Courses', icon: BookOpen, href: '/courses', roles: ['admin', 'mentor'] },
  { name: 'Classes', icon: Calendar, href: '/classes', roles: ['admin'] },
  { name: 'Attendance', icon: CheckCircle, href: '/attendance', roles: ['admin', 'mentor', 'student'] },
  { name: 'Face Enrollment', icon: Camera, href: '/face-enrollment', roles: ['student'] },
  { name: 'Schedule', icon: Clock, href: '/schedule', roles: ['admin', 'mentor', 'student'] },
];

// Filter menu items based on user role
const filteredItems = menuItems.filter(item =>
  item.roles.includes(user?.role || '')
);
```

Listing 8: Dynamic Menu Implementation

The menu system provides:

- Role-based visibility of menu items
- Active state highlighting for current page
- Collapsible sidebar for mobile views
- Icon-based navigation with tooltips

8.2 Authentication & Authorization

8.2.1 JWT-Based Authentication

```python
from jose import jwt
from datetime import datetime, timedelta

class AuthService:
    SECRET_KEY = os.getenv("JWT_SECRET_KEY")
    ALGORITHM = "HS256"
    ACCESS_TOKEN_EXPIRE_MINUTES = 30

    def create_access_token(self, user_id: str, role: str) -> str:
        expire = datetime.utcnow() + timedelta(minutes=self.ACCESS_TOKEN_EXPIRE_MINUTES)
        payload = {
            "sub": user_id,
            "role": role,
            "exp": expire
        }
        return jwt.encode(payload, self.SECRET_KEY, algorithm=self.ALGORITHM)

    def decode_token(self, token: str) -> dict:
        return jwt.decode(token, self.SECRET_KEY, algorithms=[self.ALGORITHM])
```

Listing 9: JWT Token Generation

8.2.2 Role-Based Access Control (RBAC)

```python
from fastapi import Depends, HTTPException

def require_role(*allowed_roles: str):
    def role_checker(current_user: User = Depends(get_current_user)):
        if current_user.role not in allowed_roles:
            raise HTTPException(
                status_code=403,
                detail="Insufficient permissions"
            )
        return current_user
    return role_checker

# Usage in routes
@router.get("/users")
def get_users(user: User = Depends(require_role("admin"))):
    return user_service.get_all()
```

Listing 10: RBAC Middleware