医院挂号管理系统的设计与实现

选题要求

设计一个医院挂号管理系统，包括挂号管理、科室管理、医生管理等功能。挂号管理负责挂号信息的录入、查询和统计；科室管理负责科室信息的添加、修改和查询；医生管理负责医生信息的添加、修改和查询。

引言

设计目的

本系统旨在提供一个全面的医院挂号管理解决方案，我将在这个系统中实现包括挂号管理、医生管理、排班管理、科室管理、病人管理、用户管理等功能。

设计要求

- 实现用户的注册、登录，以及用户信息的查询、修改、删除
- 实现医生信息的录入、查询、修改、删除
- 实现科室信息的录入、查询、修改、删除
- 实现病人信息的创建、查询、修改、删除
- 实现挂号信息的录入、查询、修改、删除
- 实现排班信息的录入、查询、修改、删除
- 实现管理员用户的创建以及管理员对系统信息的管理功能

设计环境

- 操作系统：Windows 10、Windwos 11
- 数据库：PostgreSQL 16.3
- 后端：Python 3.11+，Flask 3.1.0
- 前端：Node.js v22.8.0，Next.js 15.1.2

团队分工

- 项目设计与管理：林隽哲
- 数据库设计：林隽哲
- 后端开发：林隽哲
- 前端开发：林隽哲

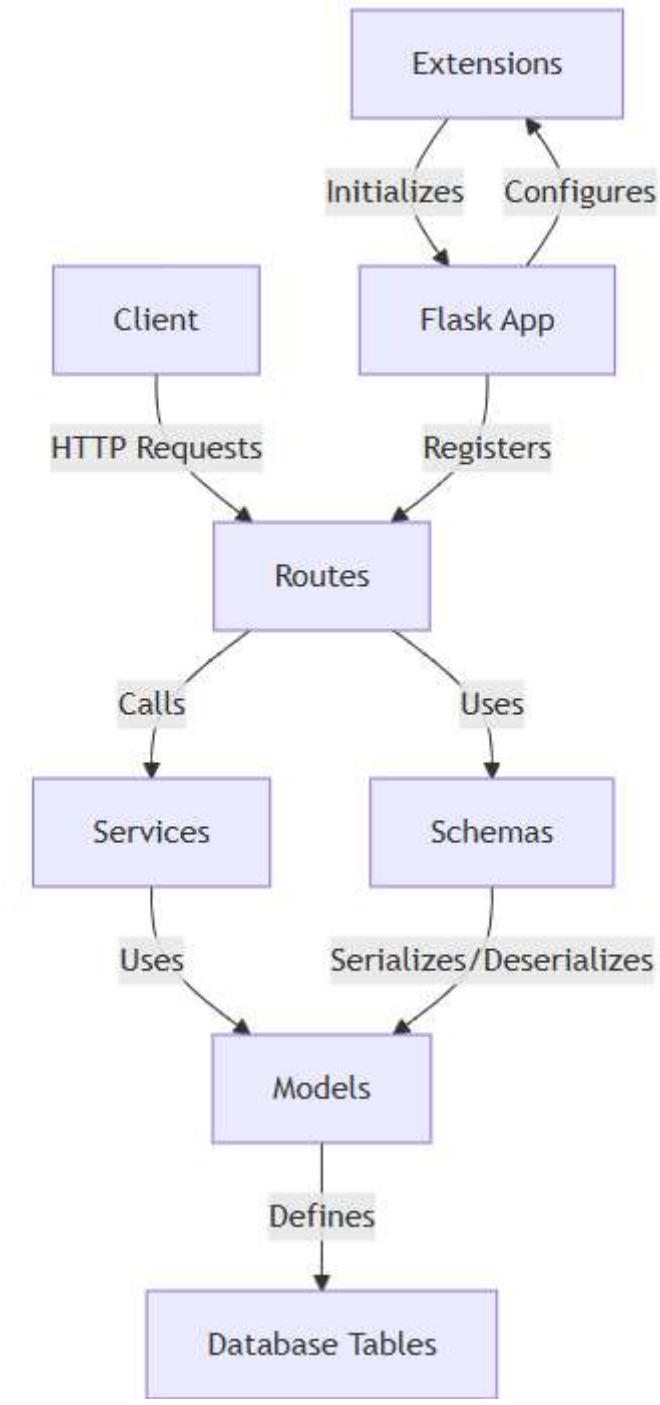
概要设计

系统需求分析

1. 用户管理：包括普通用户和管理员的注册、登录和权限控制
2. 患者管理：患者信息的CRUD操作。（设计之初我是打算直接将用户的信息作为患者信息进行处理的。但在我参考了广州一些医院官方公众号的挂号系统的实现后决定将患者信息与用户信息区分开来。在现在的实现中患者的信息将会由用户额外进行创建，并且一个用户能够创建多个患者信息。）
3. 医生管理：医生信息的CRUD操作
4. 科室管理：科室信息的CRUD操作
5. 排班管理：医生排班信息的CRUD操作
6. 挂号管理：挂号信息的CRUD操作
7. 权限控制：区分为普通用户与管理员用户（虽然起初很想实现RBAC模式来区分用户、医生与管理员角色，但由于时间不足最终改为了只有普通用户与管理员用户。现在的实现中医生信息、科室信息等均由管理员用户进行管理。）

系统架构设计

本系统采用前后端分离的架构，后端使用Flask框架提供RESTful API，前端使用Next.js构建用户界面。系统整体的架构如下（这里主要给出后端的架构）：



功能模块设计

后端功能模块

后端采用Flask框架，主要包含以下组件：

```
backend
├── app
│   ├── routes
│   │   ├── __init__.py
│   │   ├── affiliation_routes.py
│   │   ├── department_routes.py
│   │   ├── doctor_routes.py
│   │   ├── pagination.py
│   │   ├── patient_routes.py
│   │   ├── registration_routes.py
│   │   ├── schedule_routes.py
│   │   └── user_routes.py
│   ├── schemas
│   │   ├── __init__.py
│   │   ├── affiliation_schemas.py
│   │   ├── department_schemas.py
│   │   ├── doctor_schemas.py
│   │   ├── pagination.py
│   │   ├── patient_schemas.py
│   │   ├── registration_schemas.py
│   │   ├── schedule_schemas.py
│   │   └── user_schemas.py
│   ├── services
│   │   ├── __init__.py
│   │   ├── affiliation_services.py
│   │   ├── department_services.py
│   │   ├── doctor_services.py
│   │   ├── patient_services.py
│   │   ├── registration_services.py
│   │   ├── schedule_services.py
│   │   └── user_services.py
│   ├── utils
│   │   ├── auth_utils.py
│   │   ├── db_utils.py
│   │   └── route_utils.py
│   ├── __init__.py
│   ├── cli.py
│   ├── config.py
│   ├── extentions.py
│   ├── logger.py
│   └── models.py
├── migrations
├── .gitignore
├── clean.py
├── draw_er.py
├── erd_from_sqlalchemy.png
├── manage.py
├── run.py
└── test.py
```

1. 路由（Routes）：定义API端点，处理HTTP请求
2. 模式（Schemas）：定义数据序列化与反序列化的规则。（主要是用于在路由返回请求结果时，将SQLAlchemy的model转换为json的格式）
3. 服务（Services）：实现业务逻辑与数据库的数据操作
4. 模型（Models）：通过ORM定义数据库表结构
5. 工具（Utils）：提供一些辅助功能

前端功能模块

前端使用Next.js框架，并采用了APP Router结构。由于组件部分我是使用shadcn ui为主，因此这里主要关注路由部分。

前端主要包含以下组件：

```
frontend
├── app
│   ├── (auth)
│   │   ├── login
│   │   │   └── page.tsx
│   │   ├── register
│   │   │   └── page.tsx
│   │   └── layout.tsx
│   ├── (main)
│   │   ├── admin
│   │   │   ├── affiliations
│   │   │   │   └── page.tsx
│   │   │   ├── appointments
│   │   │   │   └── page.tsx
│   │   │   ├── departments
│   │   │   │   └── page.tsx
│   │   │   ├── doctors
│   │   │   │   └── page.tsx
│   │   │   ├── patients
│   │   │   │   └── page.tsx
│   │   │   ├── schedules
│   │   │   │   └── page.tsx
│   │   │   ├── users
│   │   │   │   └── page.tsx
│   │   │   ├── layout.tsx
│   │   │   └── page.tsx
│   │   ├── appointments
│   │   │   ├── [id]
│   │   │   └── page.tsx
│   │   ├── book
│   │   │   └── [scheduleId]
│   │   ├── departments
│   │   │   ├── [id]
│   │   │   └── page.tsx
│   │   ├── doctors
│   │   │   ├── [id]
│   │   │   └── page.tsx
│   │   ├── patients
│   │   │   ├── [id]
│   │   │   ├── new
│   │   │   │   └── page.tsx
│   │   │   └── page.tsx
│   │   ├── profile
│   │   │   └── page.tsx
│   │   └── layout.tsx
│   ├── globals.css
│   ├── layout.tsx
│   └── page.tsx
├── lib
│   ├── api.ts
│   ├── types.ts
│   └── utils.ts
```

Next.js使用基于文件系统的路由，这意味这文件结构将会直接映射到URL结构中。其中：

1. 用户认证模块

- 登录（ `/(auth)/login` ）
- 注册（ `/(auth)/register` ）

2. 管理员模块（ `/(main)/admin` ）

- 用户管理（ `/users` ）
- 医生管理（ `/doctors` ）
- 患者管理（ `/patients` ）
- 科室管理（ `/departments` ）

- 排班管理（ `/schedules` ）
- 挂号管理（ `/appointments` ）
- 医生-科室关联管理（ `/affiliations` ）

3. 医生模块（ `/(main)/doctors` ）

- 医生列表
- 医生详情（ `/[id]` ）

4. 科室模块（ `/(main)/departments` ）

- 科室列表
- 科室详情（ `/[id]` ）

5. 挂号模块（ `/(main)/appointments` ）

- 挂号列表
- 挂号详情（ `/[id]` ）

6. 患者模块（ `/(main)/patients` ）

- 患者列表
- 患者详情（ `/[id]` ）
- 新增患者（ `/new` ）

7. 预约挂号（ `/(main)/book/[scheduleId]` ）
8. 用户个人中心（ `/(main)/profile` ）

除此之外还值得一提的是：

- `api.ts`： 这个文件封装了与后端API的所有通信逻辑
- `types.ts`： 这个文件定义了从后端中接受的数据的类型

详细设计

Flask插件介绍（Extentions）

这里简单的介绍一下我在后端的Flask中使用到的扩展插件：

1. **Flask-SQLAlchemy**

- 提供了SQLAlchemy的支持，用于数据库的ORM开发

2. **Flask-Migrate**

- 提供了数据库迁移支持，方便管理数据库架构的变更

3. **Flask-RESTX**

- 用于构建RESTful API，提供自动生成Swagger文档的功能

4. **Flask-JWT-Extended**

- 提供JWT（JSON Web Token）支持，用于用户认证

5. **Flask-APScheduler**

- 提供任务调度功能，可以用于定时任务

6. **Flask-CORS**

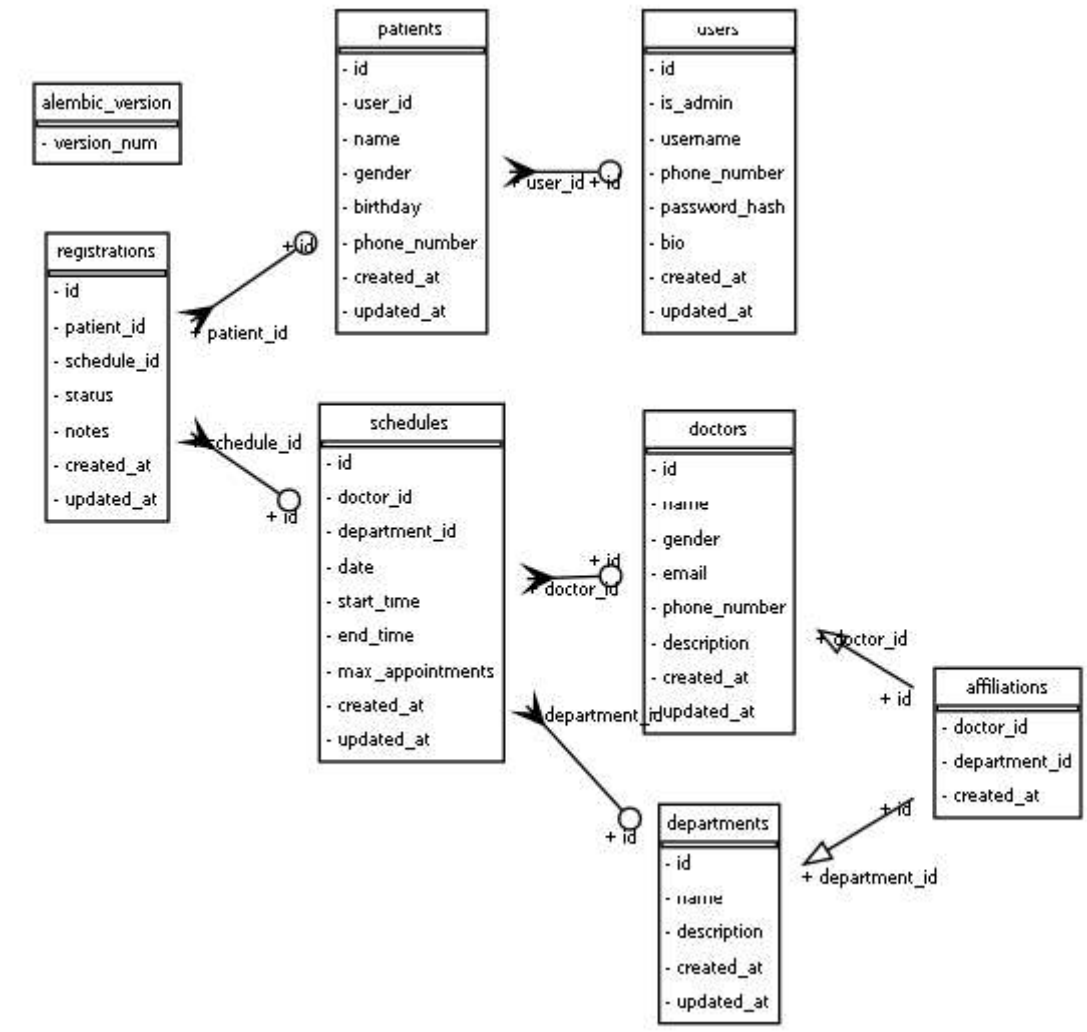
- 处理跨域资源共享（CORS），允许前端应用从不同域名访问API

7. **Waitress**（虽然严格上来说它并不属于Flask插件）

- 作为生产环境的WSGI，用于部署Flask应用。它有跨平台的支持，可以在Windows与Unix系统上运行。并且它支持多线程，可以处理并发请求

数据库设计（Models）

首先给出数据库的ER图：



本系统使用SQLAlchemy ORM（对象关系映射）来定义和管理数据库模型。ORM是一种编程技术，它让我们可以使用面向对象编程语言来操作关系数据库。同时，ORM的开发也能够有效防止SQL注入攻击，为数据库提供了保护。在我们的系统中，每个Python类代表一个数据库表，类的属性代表一个数据库表，类的属性对应表的列。

本系统中涉及到的模型如下：

1. **User（用户）**
   - 属性：id, is_admin, username, phone_number, password_hash, bio, created_at, updated_at
   - 关系：一对多关系到Patient

实现如下：

```
class User(db.Model):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    is_admin = Column(Boolean, default=False, nullable=False)
    username = Column(String(255), unique=True, nullable=False)
    phone_number = Column(String(255), unique=True, nullable=False)
    password_hash = Column(String(255))
    bio = Column(Text)
    created_at = Column(DateTime, default=datetime.now(timezone.utc))
    updated_at = Column(DateTime, default=datetime.now(timezone.utc), onupdate=datetime.now(timezone.utc))

    patients = relationship('Patient', backref='user', lazy='dynamic', cascade='all, delete-orphan')

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)
```

2. **Patient（患者）**
   - 属性：id, user_id, name, gender, birthday, phone_number, created_at, updated_at
   - 关系：多对一关系到User，一对多关系到Registration

实现如下：

```
class Patient(db.Model):
    __tablename__ = 'patients'
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('users.id', ondelete='CASCADE'), nullable=False)
    name = Column(String(255), nullable=False)
    gender = Column(Enum('male', 'female', name='gender_type'), nullable=False)
    birthday = Column(Date, nullable=False)
    phone_number = Column(String(255), unique=True, nullable=False)
    created_at = Column(DateTime, default=datetime.now(timezone.utc))
    updated_at = Column(DateTime, default=datetime.now(timezone.utc), onupdate=datetime.now(timezone.utc))

    registrations = relationship('Registration', backref='patient', lazy='dynamic', cascade='all, delete-orphan')
```

3. **Doctor（医生）**
   - 属性：id, name, gender, email, phone_number, description, created_at, updated_at
   - 关系：多对多关系到Department，一对多关系到Schedule

实现如下：

```
class Doctor(db.Model):
    __tablename__ = 'doctors'
    id = Column(Integer, primary_key=True)
    name = Column(String(255), nullable=False)
    gender = Column(Enum('male', 'female', name='gender_type'), nullable=False)
    email = Column(String(255), nullable=False)
    phone_number = Column(String(255), unique=True, nullable=False)
    description = Column(Text)
    created_at = Column(DateTime, default=datetime.now(timezone.utc))
    updated_at = Column(DateTime, default=datetime.now(timezone.utc), onupdate=datetime.now(timezone.utc))

    departments = relationship('Department', secondary=affiliations, back_populates='doctors')
    schedules = relationship('Schedule', backref='doctor', lazy='dynamic', cascade='all, delete-orphan')
```

4. **Department（科室）**
   - 属性：id, name, description, created_at, updated_at
   - 关系：多对多关系到Doctor，一对多关系到Schedule

实现如下：

```
class Department(db.Model):
    __tablename__ = 'departments'
    id = Column(Integer, primary_key=True)
    name = Column(String(255), nullable=False, unique=True)
    description = Column(Text)
    created_at = Column(DateTime, default=datetime.now(timezone.utc))
    updated_at = Column(DateTime, default=datetime.now(timezone.utc), onupdate=datetime.now(timezone.utc))

    doctors = relationship('Doctor', secondary=affiliations, back_populates='departments')
    schedules = relationship('Schedule', backref='department', lazy='dynamic', cascade='all, delete-orphan')
```

5. **Schedue（排班）**
   - 属性：id, doctor_id, department_id, date, start_time, end_time, max_appointments, created_at, updated_at
   - 关系：多对一关系到Doctor和Department，一对多关系到Registration

实现如下：

```
class Schedule(db.Model):
    __tablename__ = 'schedules'
    id = Column(Integer, primary_key=True)
    doctor_id = Column(Integer, ForeignKey('doctors.id', ondelete='CASCADE'))
    department_id = Column(Integer, ForeignKey('departments.id', ondelete='CASCADE'))
    date = Column(Date, nullable=False)
    start_time = Column(Time, nullable=False)
    end_time = Column(Time, nullable=False)
    max_appointments = Column(Integer, nullable=False)
    created_at = Column(DateTime, default=datetime.now(timezone.utc))
    updated_at = Column(DateTime, default=datetime.now(timezone.utc), onupdate=datetime.now(timezone.utc))

    registrations = relationship('Registration', backref='schedule', lazy='dynamic', cascade='all, delete-orphan')

    @property
    def available_slots(self):
        taken_slots = self.registrations.filter(Registration.status != 'cancelled').count()
        return max(0, self.max_appointments - taken_slots)
```

6. **Registration（挂号）**
   - 属性：id, patient_id, schedule_id, status, notes, created_at, updated_at
   - 关系：多对一关系到Patient和Schedule

实现如下：

```
class Registration(db.Model):
    __tablename__ = 'registrations'
    id = Column(Integer, primary_key=True)
    patient_id = Column(Integer, ForeignKey('patients.id', ondelete='CASCADE'), nullable=False)
    schedule_id = Column(Integer, ForeignKey('schedules.id', ondelete='CASCADE'), nullable=False)
    status = Column(Enum('scheduled', 'completed', 'cancelled', name='registration_status'), default='scheduled', nullable=False)
    notes = Column(Text)
    created_at = Column(DateTime, default=datetime.now(timezone.utc))
    updated_at = Column(DateTime, default=datetime.now(timezone.utc), onupdate=datetime.now(timezone.utc))
```

7. **Affiliation（医生-科室关联：隶属关系）**
   - 属性：doctor_id, department_id, created_at
   - (严格上来说这并不是模型，是多对多关系表)

实现如下：

```
affiliations = db.Table('affiliations',
    Column('doctor_id', Integer, ForeignKey('doctors.id', ondelete='CASCADE'), primary_key=True),
    Column('department_id', Integer, ForeignKey('departments.id', ondelete='CASCADE'), primary_key=True),
    Column('created_at', DateTime, default=datetime.now(timezone.utc))
)
```

服务层（Services）设计

服务层（Services）是本系统架构中的关键组件。它负责实现业务逻辑，并通过使用ORM与数据库进行交互，处理复杂的查询和数据操作。

下面是一个服务层函数的示例，展示了用户搜索功能的实现：

```
def search_users(query, page=1, per_page=10, sort='id', reverse=False):
    if not hasattr(User, sort):
        raise ValueError(User, sort)

    search = f"%{query}%"
    query = User.query.filter(
        (User.username.ilike(search)) | (User.phone_number.ilike(search))
    )
    order = desc(getattr(User, sort)) if reverse else getattr(User, sort)
    query = query.order_by(order)

    pagination = query.paginate(page=page, per_page=per_page, error_out=False)
    return pagination.items, pagination.total, pagination.pages > page
```

数据模式（Schemas）设计

在本系统中，数据模式（Schemas）定义了API的输入和输出结构。在本系统中我基于Flask-RESTX对数据模式进行了定义。Flask-RESTX同时还会提供一个API文档自动生成的功能，非常方便，具体可以看后续的API设计。

以下是用户相关的两个模式定义示例：

```
user_model = api.model('User', {
    'id': fields.Integer(readonly=True, description='The user unique identifier'),
    'is_admin': fields.Boolean(description='Whether the user is an admin'),
    'username': fields.String(required=True, description='The user username'),
    'phone_number': fields.String(required=True, description='The user phone number'),
    'password_hash': fields.String(required=True, description='The hashed password', attribute='password_hash'),
    'bio': fields.String(description='The user bio'),
    'created_at': fields.DateTime(readonly=True, description='Creation date'),
    'updated_at': fields.DateTime(readonly=True, description='Last update date')
})

user_create_model = api.model('UserCreate', {
    'username': fields.String(required=True, description='The user username'),
    'phone_number': fields.String(required=True, description='The user phone number'),
    'password': fields.String(required=True, description='The user password'),
    'bio': fields.String(description='The user bio')
})
```

API设计（Routes）

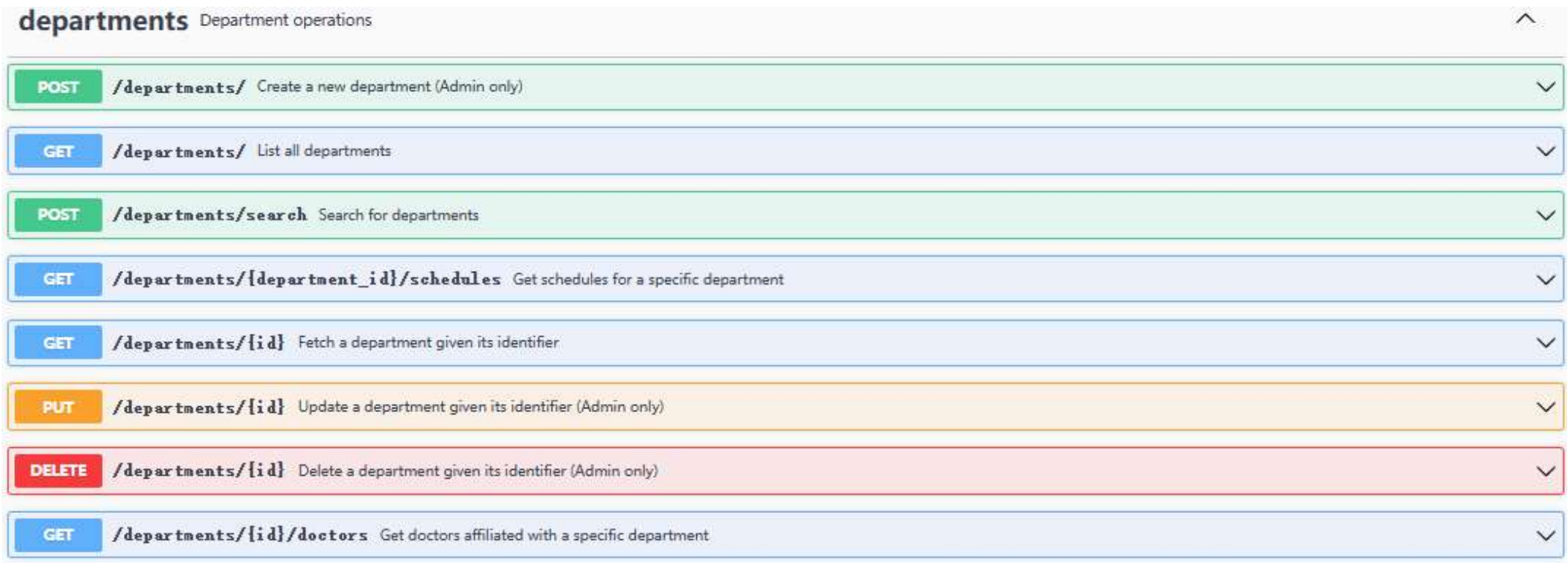
本系统提供RESTful风格的API，主要包括以下端点：

- `\users`： 用户相关操作

- `\patients`： 患者相关操作

- `\doctors`： 医生相关操作

- `\departments`： 科室相关操作



- `\schedules`： 排班相关操作

- `\registrations`： 挂号相关操作

- `\affiliations`： 医生-科室关系（隶属关系）操作

前端API接口设计

前端的路由实现实际上在概要设计中已经可以了解得七七八八了，这里主要关注前端在API接口上的设计。

在前端部分，我编写了一个 `lib/api.ts` 的接口文件，用于处理前端与后端的所有通信。

首先我先封装了一个URL的请求函数，它可以简单的帮我过滤出请求成功的response并从中提取出json数据：

```
async function fetchWithErrorHandling(url: string, options: RequestInit = {}): Promise<any | null> {
  const response = await fetch(url, options);
  if (!response.ok) {
    const errorMessage = `HTTP error! status: ${response.status}`;
    throw new Error(errorMessage);
  }

  try {
    return await response.json();
  } catch {
    return null;
  }
}
```

然后下面是几个接口的示例：

```
export const api = {
  login: async (username: string, password: string): Promise<{ access_token: string; user: User }> => {
    return fetchWithErrorHandling(`${BASE_URL}/users/login`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ username, password }),
    });
  },

  getCurrentUser: async (token: string): Promise<User> => {
    return fetchWithErrorHandling(`${BASE_URL}/users/me`, {
      headers: { 'Authorization': `Bearer ${token}` },
    });
  },
}
```

`api.login` 会将 `username` 与 `password` 打包为json格式发送到后端，后端接收请求判断登录成功后将会返回一个令牌。只需要通过将令牌 `token` 写入到 `Authorization` 字段后再向后端发送请求，后端便可判断出请求来自哪一个用户，并根据该用户提供服务。

调试与运行结果

运行后端

1. 创建一个新的虚拟环境

```
[K] 754.42ms backend $ conda create -n hospital python=3.11
Channels:
 - defaults
Platform: win-64
Collecting package metadata (repodata.json): done
Solving environment: done
```

2. 通过 `requirements.txt` 安装依赖

```
# requirements.txt

Faker==30.8.2
Flask==3.0.3
Flask-APScheduler==1.13.1
Flask-Caching==2.3.0
Flask-Cors==5.0.0
Flask-JWT-Extended==4.7.1
Flask-Login==0.6.3
Flask-Migrate==4.0.7
flask-restx==1.3.0
Flask-SQLAlchemy==3.1.1
SQLAlchemy==2.0.36
sqlalchemy_schemadisplay==2.0
waitress==3.0.2
Werkzeug==3.0.4
psycopg2==2.9.10
```

```
[K] 13490.87ms backend $ conda activate hospital
(hospital) [K] 803.58ms backend $ pip install -r .\requirements.txt
Collecting Faker==30.8.2 (from -r .\requirements.txt (line 1))
  Using cached Faker-30.8.2-py3-none-any.whl.metadata (15 kB)
Collecting Flask==3.0.3 (from -r .\requirements.txt (line 2))
  Using cached flask-3.0.3-py3-none-any.whl.metadata (3.2 kB)
Collecting Flask-APScheduler==1.13.1 (from -r .\requirements.txt (line 3))
```

3. 初始化环境变量

```
# .env

# Flask environment
FLASK_ENV=development

# Database URL
DATABASE_URL=postgresql://postgres:postgres@localhost:5432/hospital

# Secret keys
SECRET_KEY=your-secret-key
JWT_SECRET_KEY=your-jwt-secret-key
```

4. 初始化数据库（这里假设你已经创建了数据库）

```
(hospital) [K] 78.3ms backend 📁 $ flask db init
Generated admin password: UQLHFcUojHAd7ybV
Creating directory 'D:\\Program\\Code\\2_Computer_Science\\DataBase\\learning\\experiment\\final\\Hospital\\backend\\migrations' ...
  done
Creating directory 'D:\\Program\\Code\\2_Computer_Science\\DataBase\\learning\\experiment\\final\\Hospital\\backend\\migrations\\ver
sions' ...  done
Generating D:\Program\Code\2_Computer_Science\DataBase\learning\experiment\final\Hospital\backend\migrations\alembic.ini ...  done
Generating D:\Program\Code\2_Computer_Science\DataBase\learning\experiment\final\Hospital\backend\migrations\env.py ...  done
Generating D:\Program\Code\2_Computer_Science\DataBase\learning\experiment\final\Hospital\backend\migrations\README ...  done
Generating D:\Program\Code\2_Computer_Science\DataBase\learning\experiment\final\Hospital\backend\migrations\script.py.mako ...  don
e
Please edit configuration/connection/logging settings in 'D:\\Program\\Code\\2_Computer_Science\\DataBase\\learning\\experiment\\fin
al\\Hospital\\backend\\migrations\\alembic.ini' before proceeding.
(hospital) [K] 1315.83ms backend 📁 $ python manage.py makemigrations
Generated admin password: IVklTXoMM5wLjFIb
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'departments'
INFO  [alembic.autogenerate.compare] Detected added table 'doctors'
INFO  [alembic.autogenerate.compare] Detected added table 'users'
INFO  [alembic.autogenerate.compare] Detected added table 'affiliations'
INFO  [alembic.autogenerate.compare] Detected added table 'patients'
INFO  [alembic.autogenerate.compare] Detected added table 'schedules'
INFO  [alembic.autogenerate.compare] Detected added table 'registrations'
Generating D:\Program\Code\2_Computer_Science\DataBase\learning\experiment\final\Hospital\backend\migrations\versions\747480f5f59b_.
py ...  done
Created a new migration.
(hospital) [K] 1416.04ms backend 📁 $ python manage.py migrate
Generated admin password: AsG5BZqtgKapVczp
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> 747480f5f59b, empty message
Applied all pending migrations.
(hospital) [K] 2399.65ms backend 📁 $ python .\test.py
Generated admin password: IsBuK4QUrE9jjnqf
Generating test data...
Test data generation complete.
(hospital) [K] 4865.11ms backend 📁 $
```

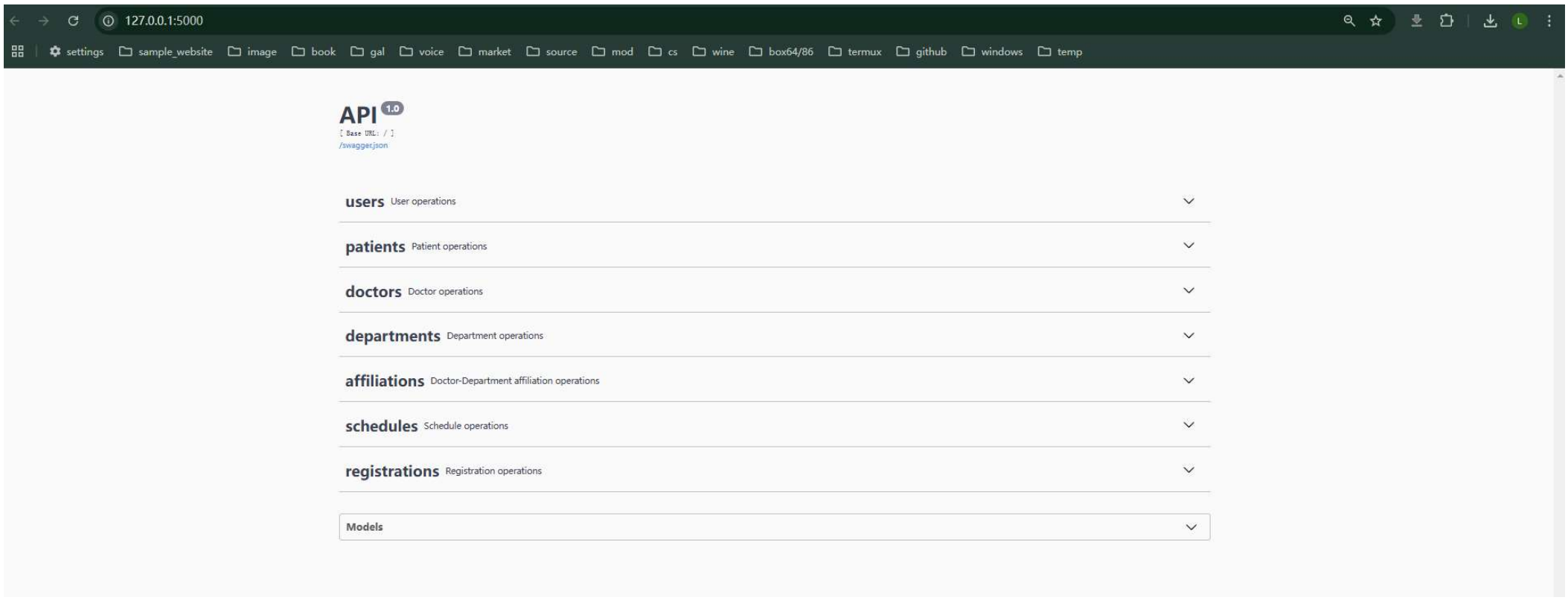
5. flask应用启动测试

```
(hospital) [K] 4865.11ms backend 📁 $ python .\run.py
Generated admin password: HU4I8gce6bIZkBEL
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://198.18.0.1:5000
Press CTRL+C to quit
```

6. 通过Waitress启动服务

```
(hospital) [K] 18675.08ms backend 📁 $ waitress-serve --host 127.0.0.1 --port 5000 --call app:create_app
Generated admin password: VEngr2nP3Kkm4mpL
INFO:waitress:Serving on http://127.0.0.1:5000
```

7. 在本地浏览器中打开查看效果



运行前端

1. 安装依赖

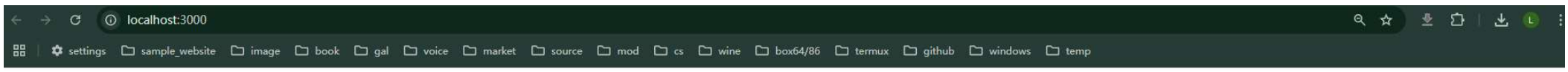
在前端项目根目录运行 `npm install` ， 这里略过

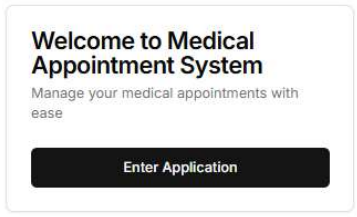
2. 运行前端

```
[K] 1497.79ms frontend 📁 $ npm run dev

> frontend@0.1.0 dev
> next dev
```

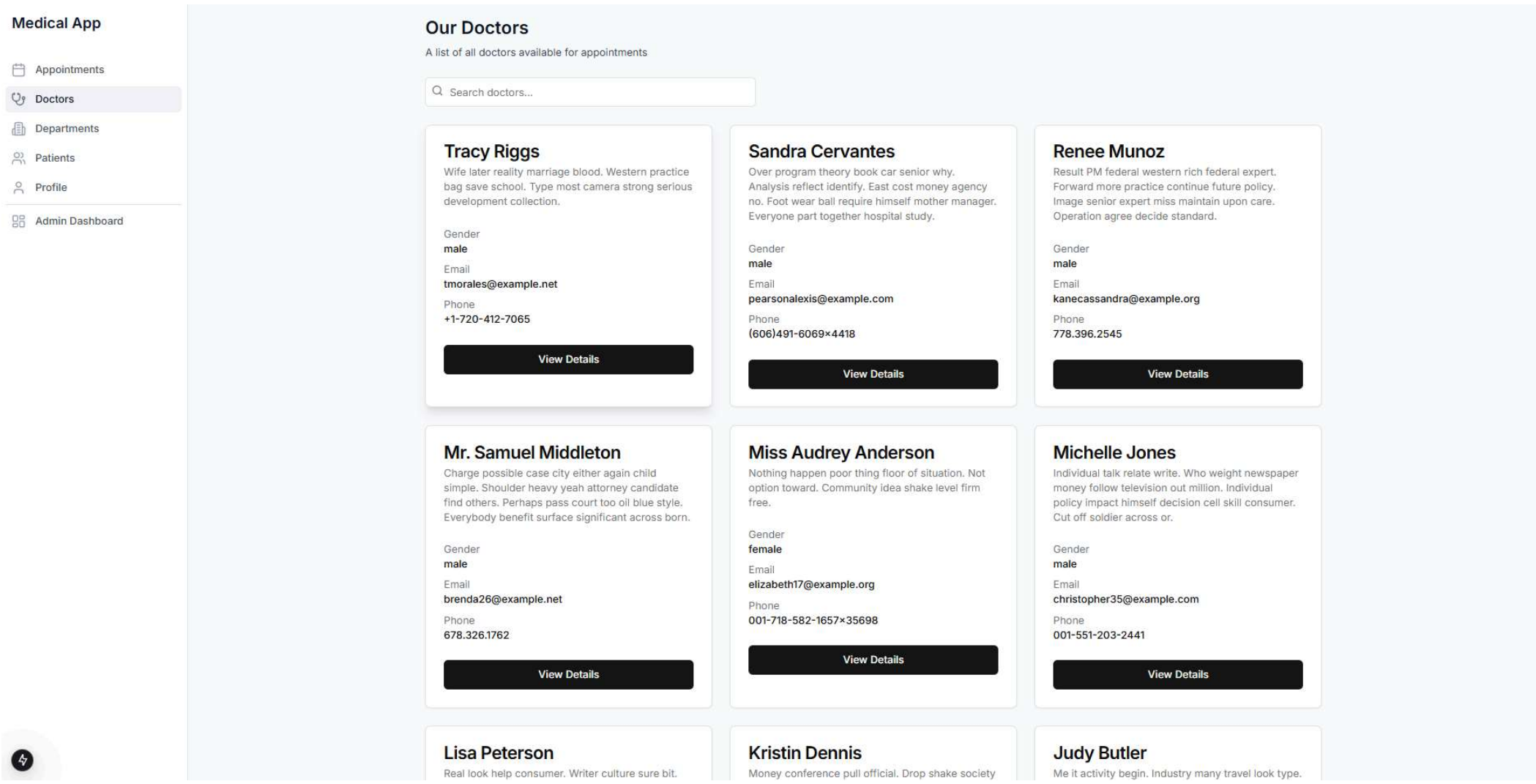
3. 在本地浏览其中打开查看效果

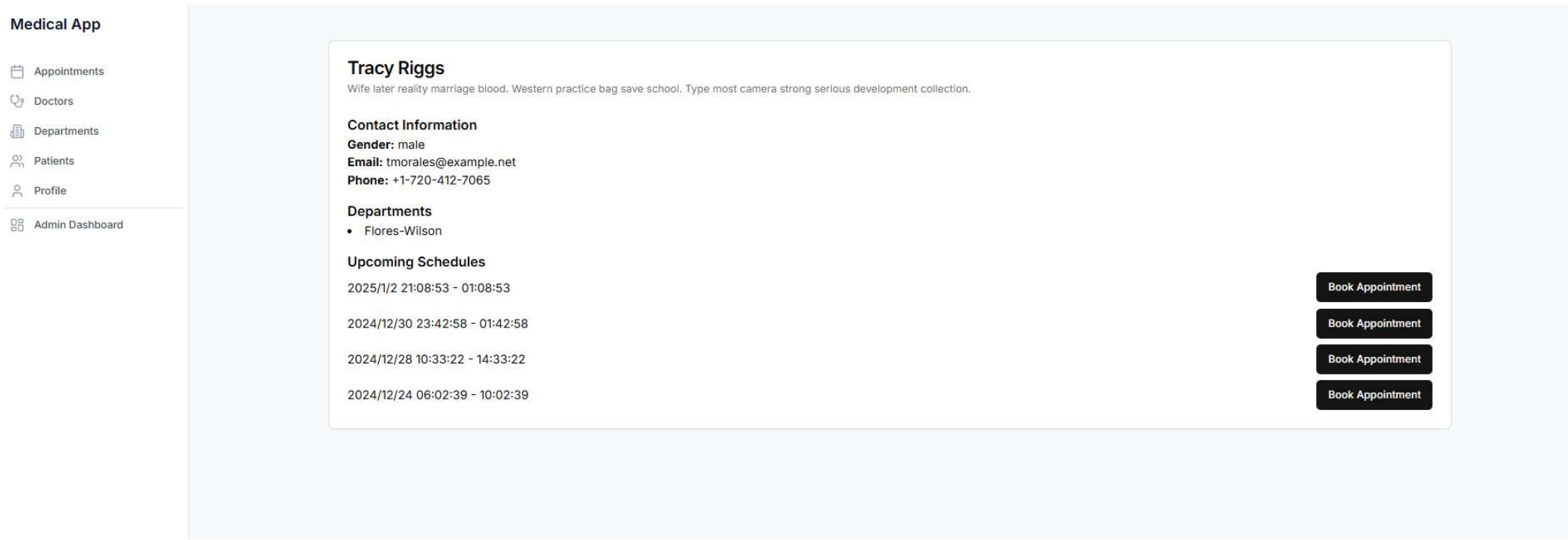




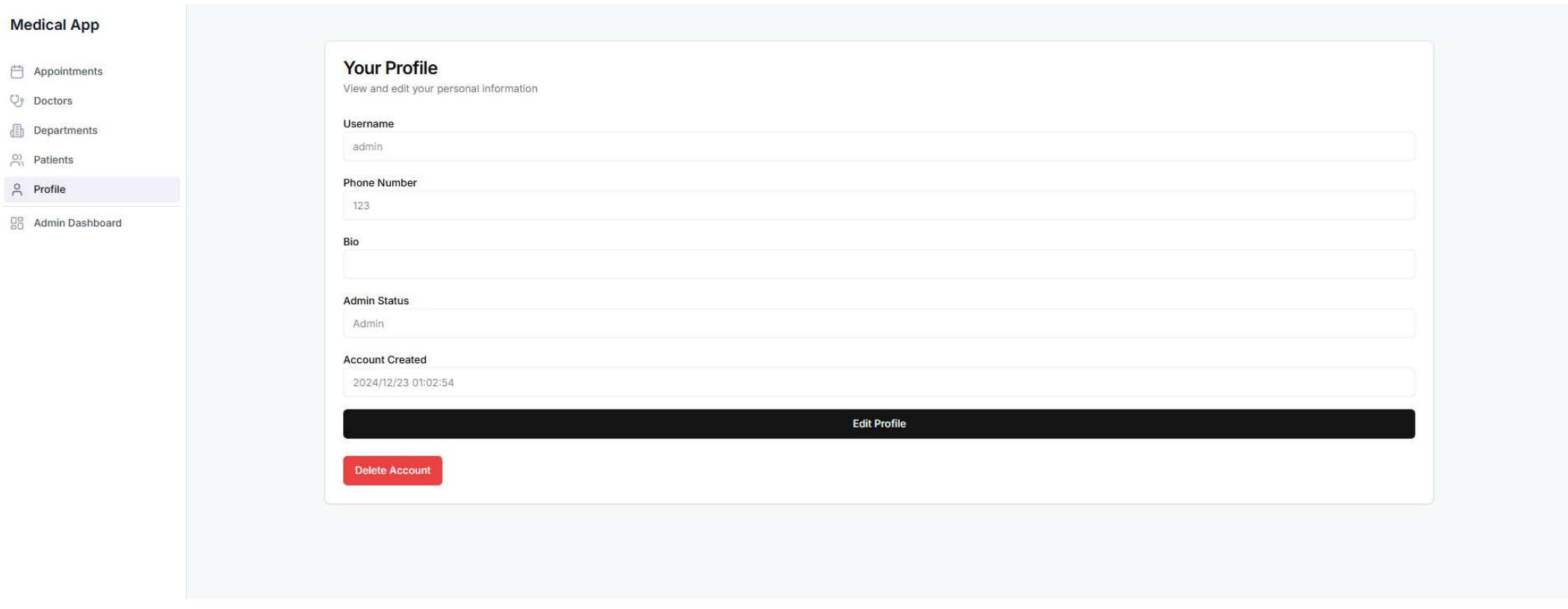
最终效果展示

- doctors页面

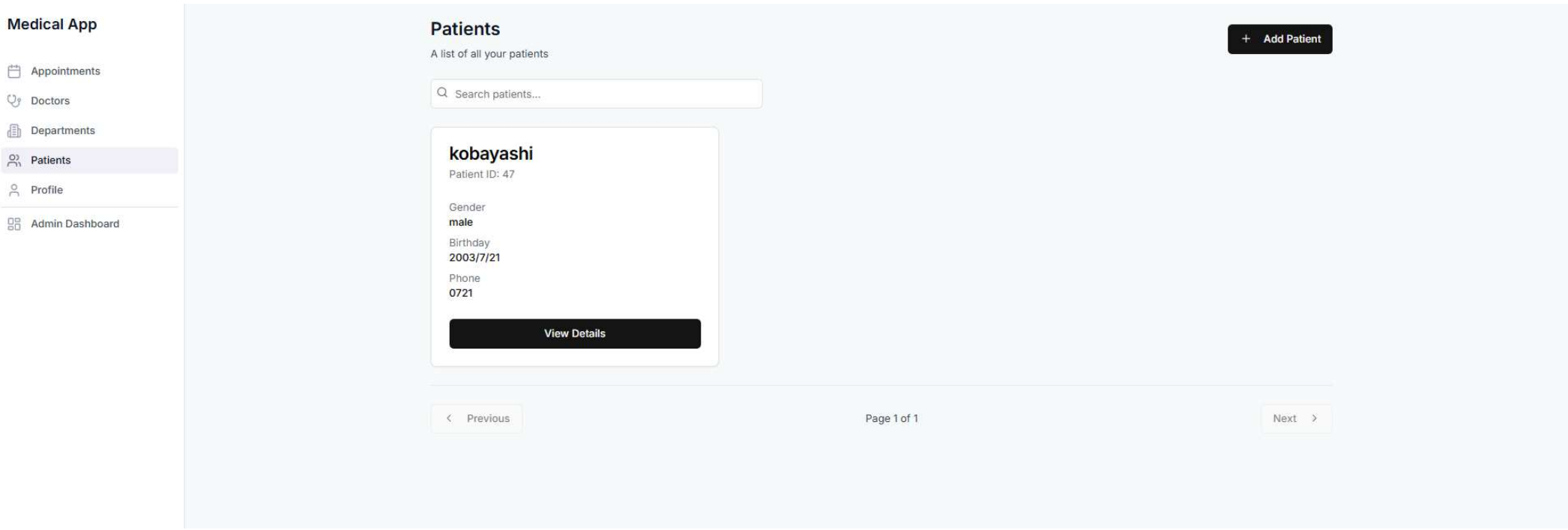




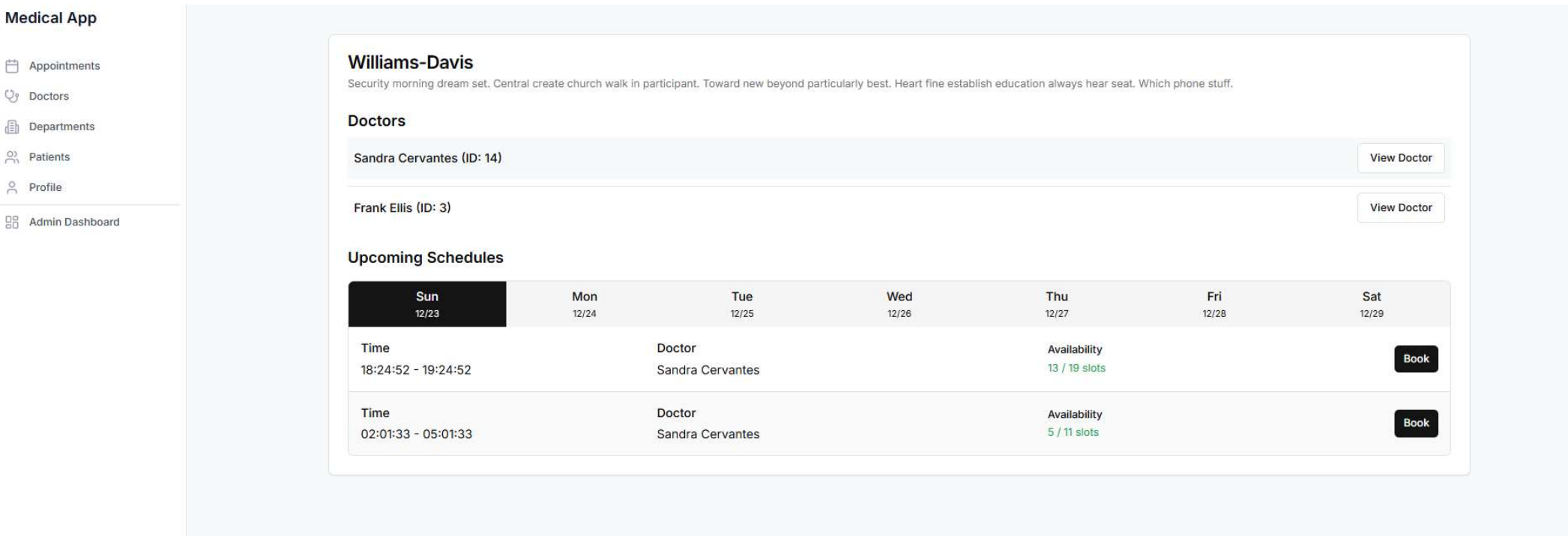
- profile页面



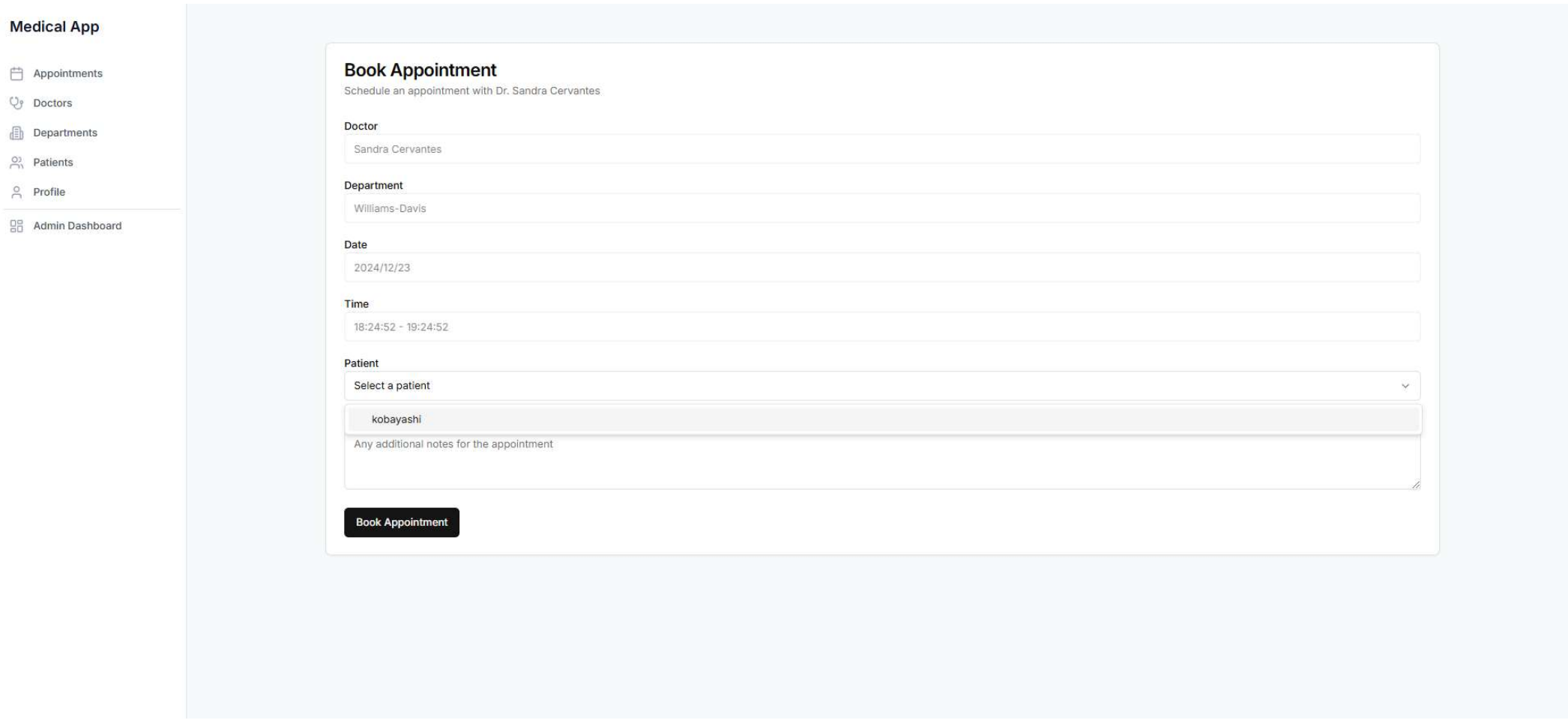
- patients页面



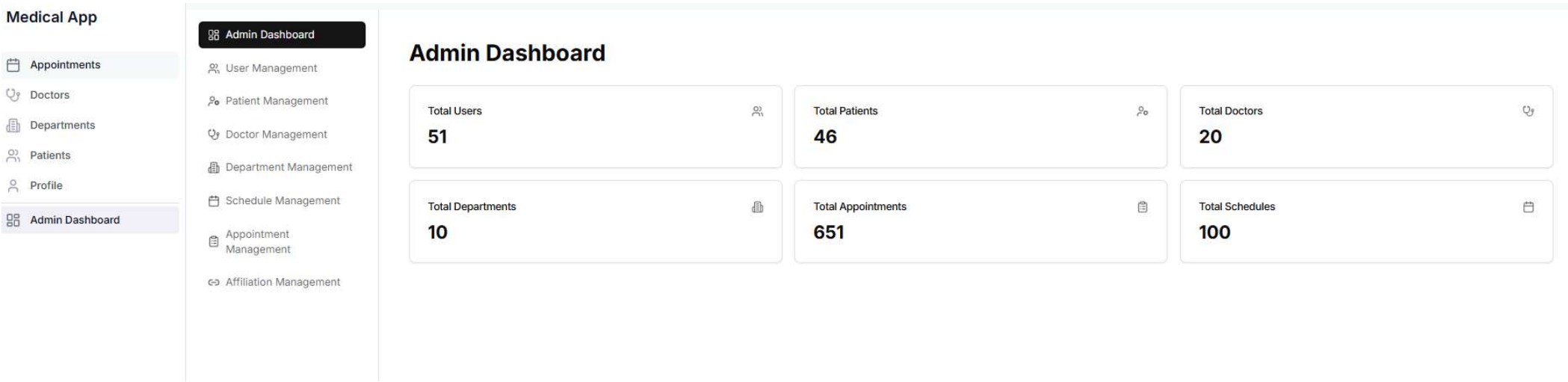
- schedule页面



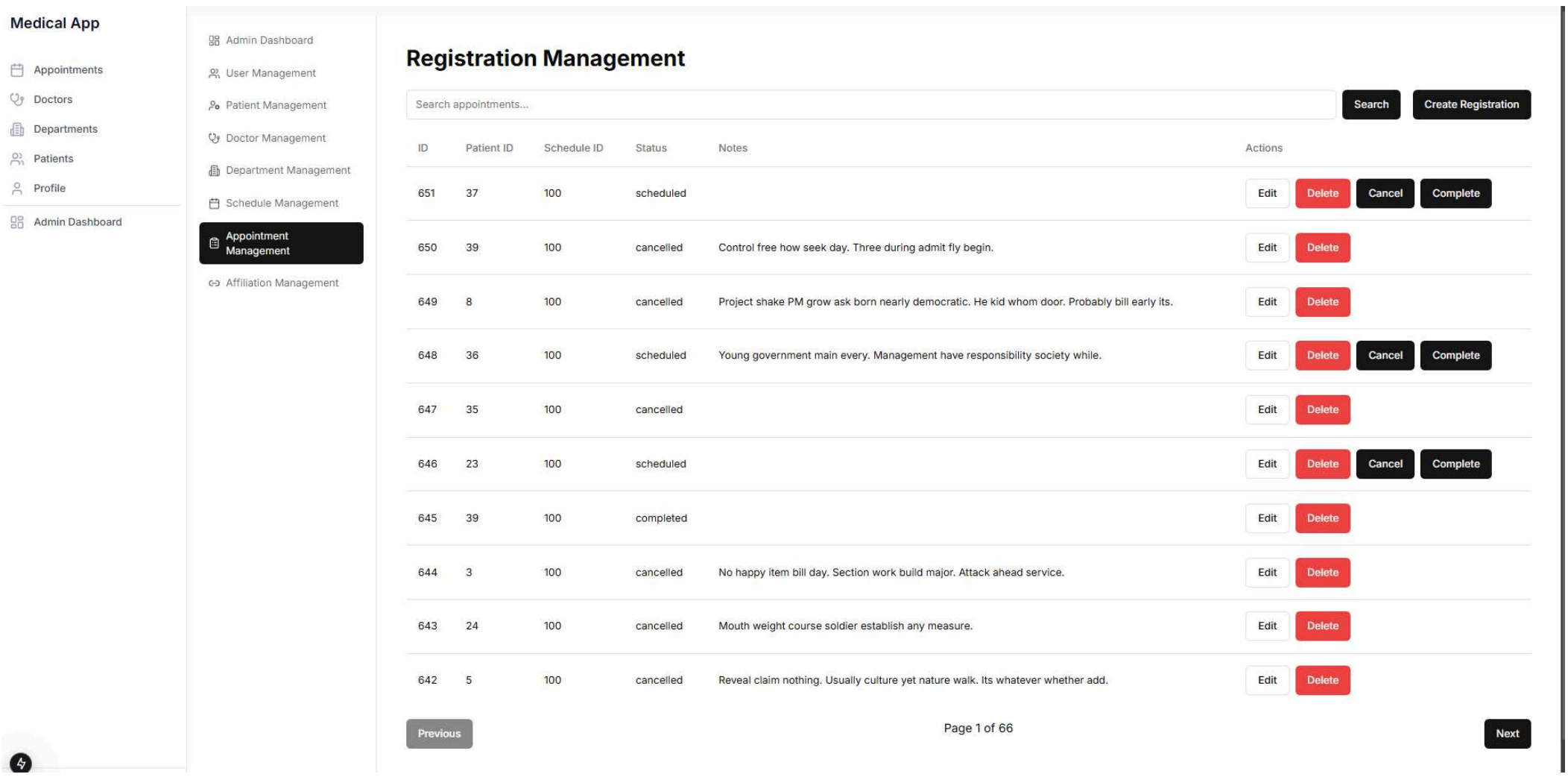
- book页面



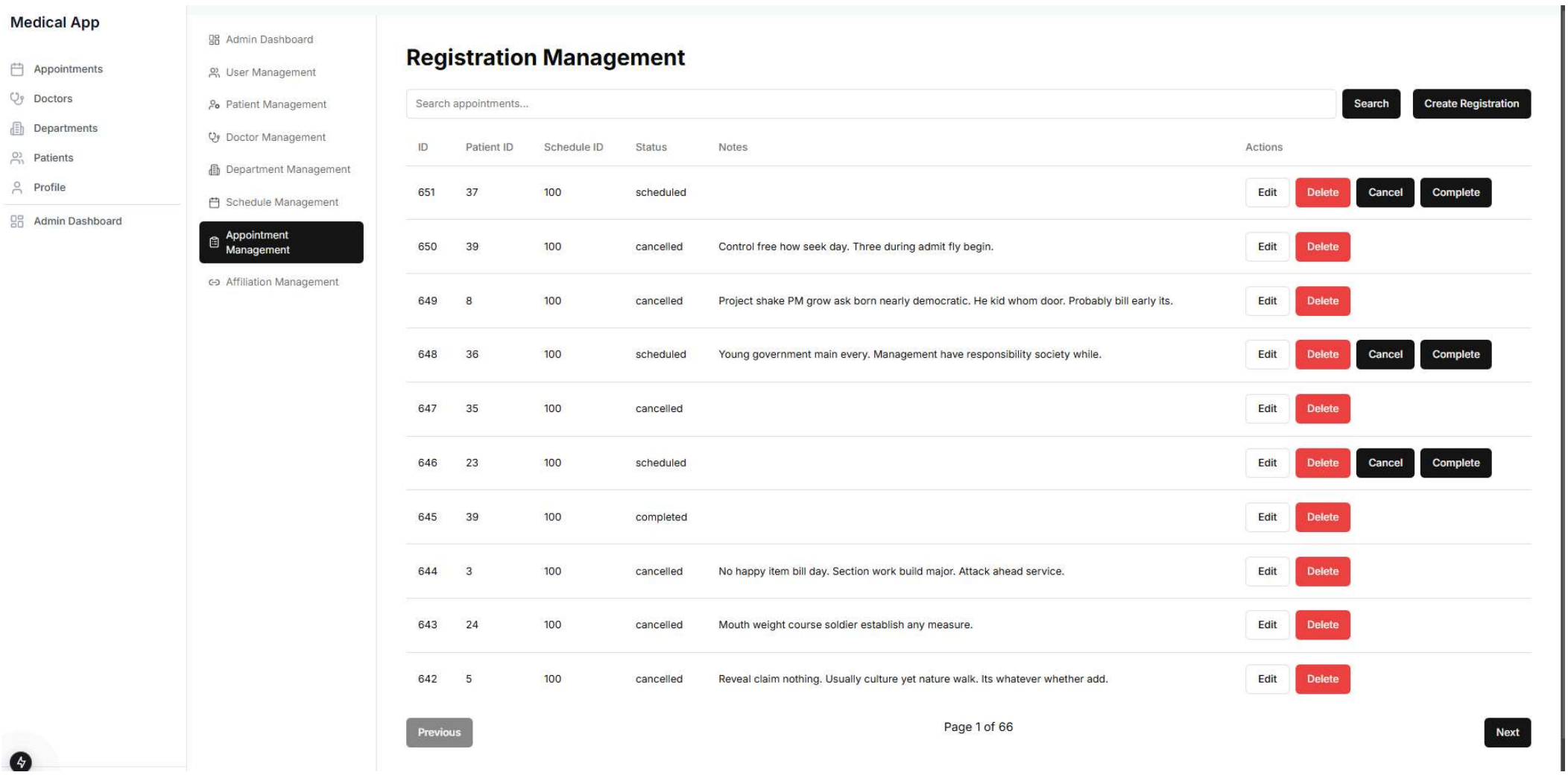
- admin dashboard页面



- admin registration management页面



- admin affiliation management页面



存在的主要问题

尽管我们的医院挂号管理系统已经实现了基本功能，但在开发过程中我们也发现了一些需要进一步改进的问题：

权限管理系统不够完善

当前系统只是简单地区分了管理员（admin）和普通用户（user）两种角色，这种粗糙的权限划分无法满足复杂医院环境中的精细化权限管理需求。在现在的系统中，所有的医生都需要通过admin用户访问系统，从数据安全和用户隐私的方面考虑这显然是不合理的安排。

缺乏有效的缓存机制

目前的系统没有实现有效的缓存机制，这可能会在以下方面造成问题：

1. 系统性能：频繁的数据库查询可能会导致系统响应速度变慢，特别是在高并发情况下
2. 用户体验：某些不常变化的数据（如科室列表、医生信息等）每次都需要重新加载，影响用户体验
3. 服务器负载：没有缓存会增加服务器的负载，可能会影响系统的稳定性

实现适当的缓存策略可以显著提高系统性能和用户体验。

前端实现较为简陋且功能不完整

由于时间与个人能力的限制，前端界面的设计较为基础。并且有一些复杂的数据管理功能等均只在后端的API中实现，前端中缺少相应的界面。

课程设计总结

在本次课程设计中，我成功开发了一个基于Flask和Next.js的医院挂号管理系统。该系统实现了用户管理、医生管理、科室管理、排班和挂号等核心功能，采用前后端分离架构。

在本次开发中我首次尝试使用Next.js框架来构建前端，其文件系统路由等特性给我带来了十分新鲜的体验。其服务端渲染（SSR）以及静态站点生成（SSG）的思想也让我感到非常有趣，而这也是我这次想使用Next.js来进行开发而不是React.js的主要原因（虽然最后因为时间关系我没怎么用到这部分，但之后我会在开发其它项目的时候学习使用它们）。

在后端的开发过程中，我也更进一步地熟悉了Flask框架的使用。Flask是我十分喜欢的一个框架，它提供了简洁的API，并且有着优秀的扩展性。这次的开发同时也锻炼了我对Flask应用的编写能力，可以说是受益良多。

总而言之，这个项目不仅让我提升了技术能力，也培养了独立解决问题的能力。这次的开发经验无疑为我之后的项目开发打下了更加坚实的基础。

参考文献

1. Flask官方文档，https://flask.palletsprojects.com/
2. SQLAlchemy官方文档，https://docs.sqlalchemy.org/
3. Next.js官方文档，https://nextjs.org/docs
4. shadcn ui官方文档，https://ui.shadcn.com/docs