Maven-Clinic Assignment Design Notes

Introduction

I would like to take the moment to Thank Maven Clinic for considering my application for employment. This document is an attempt to explain my thought process behind some of the design decisions I have made while implementing my solution.

Problem Statement

The ask is the design a service to expose 2 APIs to implement a basic appointment scheduling system.

The service has these requirements:

- GET API to get all appointments for a given patient
- POST API to create a new appointment

With the following constraints.

- All appointments must start and end on the hour or half hour
- All appointments are exactly 30 minutes long
- A user can only have 1 appointment on a calendar date

My Design Choices

- All patients and practitioner are operating in UTC Time (I mean literally :))

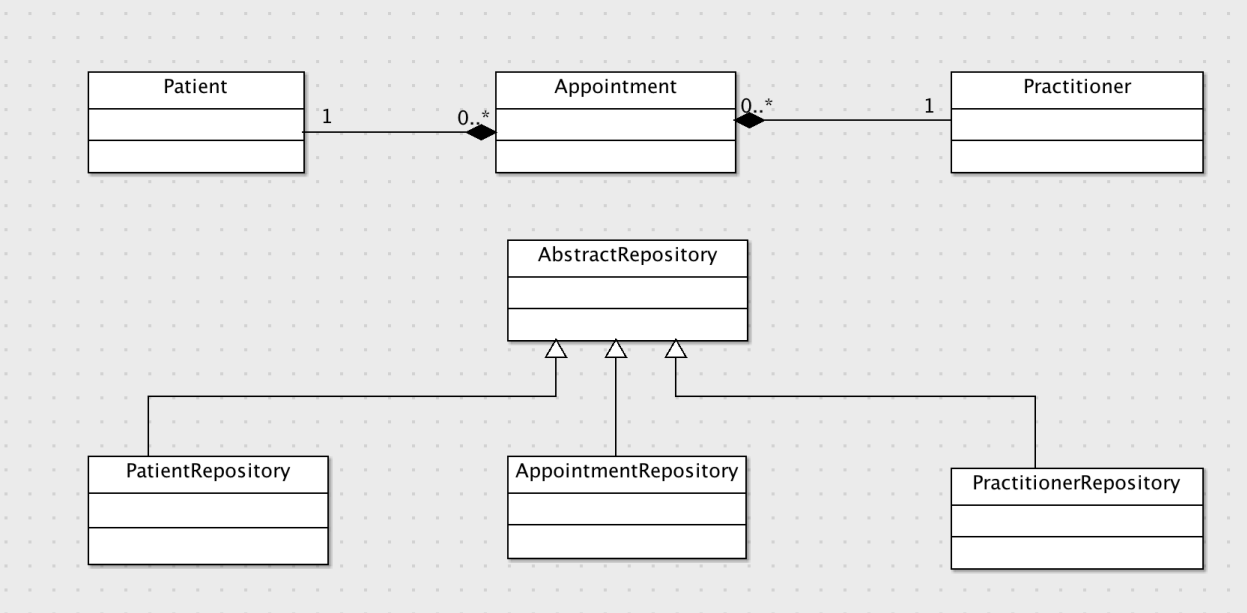
- Although the choice of language was left open, I have chosen Python as it was mentioned that its the language of choice at Maven-Clinic for backend services.

- Since it was left pretty open on the choice of the data store, I chose SQLite as its drivers come bundled with Python 3 without the need to run a dedicated service unlike MySQL/ Postgres.

- I chose an ORM (SQLAlchemy) over using plain SQL as queries using SQLAlchemy could easily be unit tested with an in memory SQLite and could scale if the implementation decided to use a more scalable db like MySQL/Postgres over time. In other words my queries would be database agnostic.

- I chose to implement the Repository pattern to enable flask routes to mock db calls and help unit test the controller logic (I could not get to testing the controller logic due to the limit on time). The repository pattern is explained in detail here https://www.cosmicpython.com/book/chapter_02_repository.html

Data Model



The Model consists of Patient, Practitioner and Appointment.
The above UML class model describes the relationship between them.

AbstractRepository helps in implementing common methods like find_by_id and create. Since AbstractRepository exposes these common methods, derived classes like PatientRepository, PractitionerRepository and AppointmentRepository inherit the functionality.

More on Repository pattern here…
https://www.cosmicpython.com/book/chapter_02_repository.html

Libraries and Frameworks I have used

- **Flask and FlaskAPI** : I have used FlaskAPI as its simple to use and is widely used to implement REST APIs in Python
- **SQLAlchemy** : An ORM greatly simplifies unit testing (see repository_test.py)
- **Flask-SQLAlchemy** : Used in conjunction with Flask to improve db session management.
- **Alembic** : DB Migration tool analogous to Java Liqubase/Flyway to version and manage schema migration

Setting up the Service

- Create a virtual environment using `mkvirtualenv -p python3 <name>`
- Run **./setup.sh**. This will run pip install and initialize SQLite db over alembic on **/tmp/mc-test.db**
- Run **./run.sh** to run the service
- Curl command to show all appointments for a patient

```
curl -X GET \
  http://127.0.0.1:5000/api/patients/1/appointments \
  -H 'cache-control: no-cache' \
  -H 'content-type: application/json’
```

- Curl command to create new appointment

```
curl -X POST \
  http://127.0.0.1:5000/api/patients/1/appointment \
  -H 'cache-control: no-cache' \
  -H 'content-type: application/json' \
  -d '{
      "practitioner_id" : 1,
      "start_time" : "2021-01-07 00:00:00"
}'
```

Unit-Testing

- Using the Repository pattern enabled me to unit test my core SQL logic by creating a SQLite db in memory and running the queries against that.
- I could not implement unit-tests on the routes due to the lack of time (more on this below).

Improvements/Things I would do if I had more time

- Unit testing the controller/Routes : This can be done by mocking the Repository and just testing the Flask routing logic
- All appointment times are taken as input in UTC and logic to check if user has no prior appointments for the day is calculated in UTC instead of taking into account the practitioner’s timezone.
- Dockerizing the application