Documentation for UniHealth Branch #1

Author Dingsu Wang & Frederick Morlock

Table of Contents

Description

Our **EHR (Electronic Health Record)** system is designed around being the patient's daily health companion. Our system enables patients to track their health daily and view notes and diagnoses from doctors. Through our application we are able to connect patients and doctors on a single unified health record platform.

Process

For our projects, we adopt the **Agile development** method. We focused on fast, incremental changes to the codebase while utilizing concurrent programming to speed development along. Using Agile development, we were able to cut down the development time of the application to only a few weeks.

In the spirit of Agile development, one of the main techniques we utilized in our development process was a method of disjoint collaborative programming. We designed our development environment in a way where the frontend and backend systems were completely isolated, allowing for concurrent development of both the backend and frontend. There one was one easy to use and extend class, named `Postman`, which served as the bridge between the frontend and the backend.

By utilizing our approach to concurrent programming, we were able to implement features on the frontend or backend without the other system being fully ready.

During our development process, there are several **issues** we faced:

- During one the iterations we had, we added a factory design pattern into our backend codes for handling login/register API for different users. However, we found it to be useless and redundant in one of the iterations later. Instead, we replaced it with a more rudimentary pattern.

- At the beginning of our development process, we simply applied unit test in our projects. After the project developed, there were an increasing number of APIs and functionalities that needed to be tested, leading to our test environment becoming complicated. We integrated the idea of regression testing in order to solve this problem of complexity.

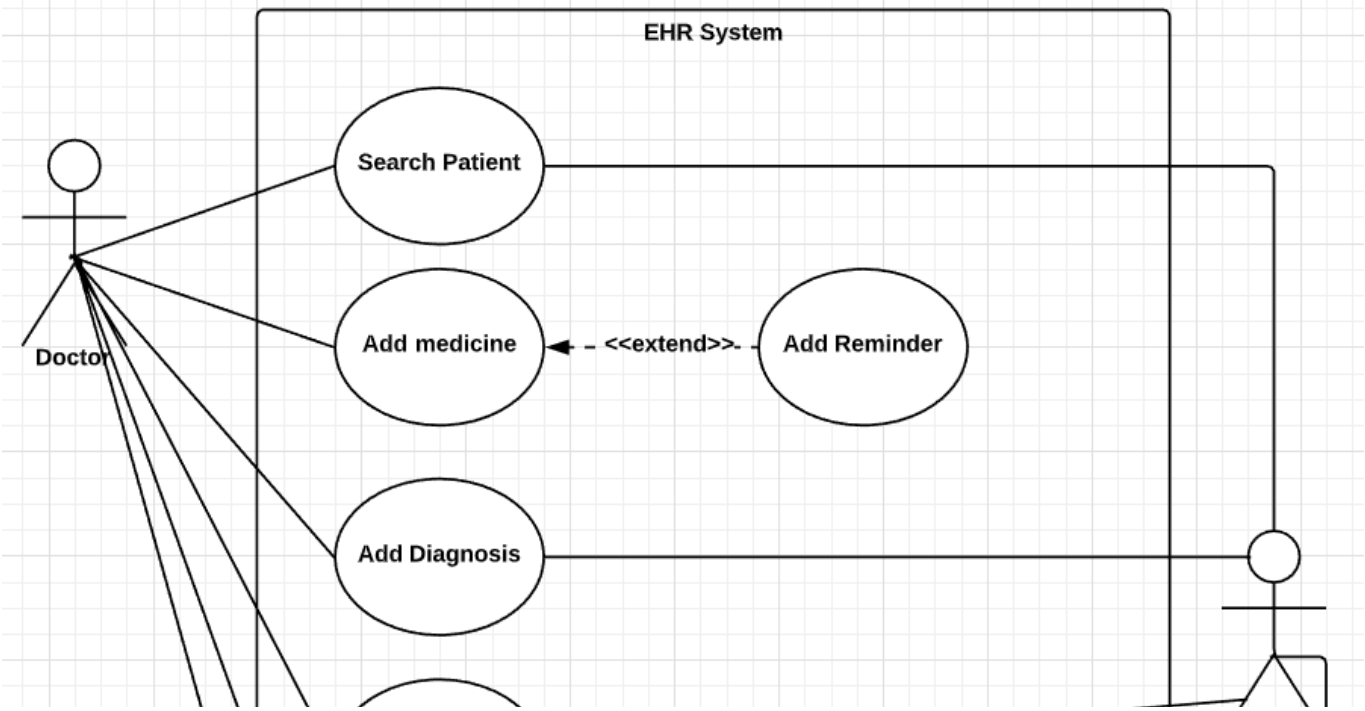
Requirements & Specifications

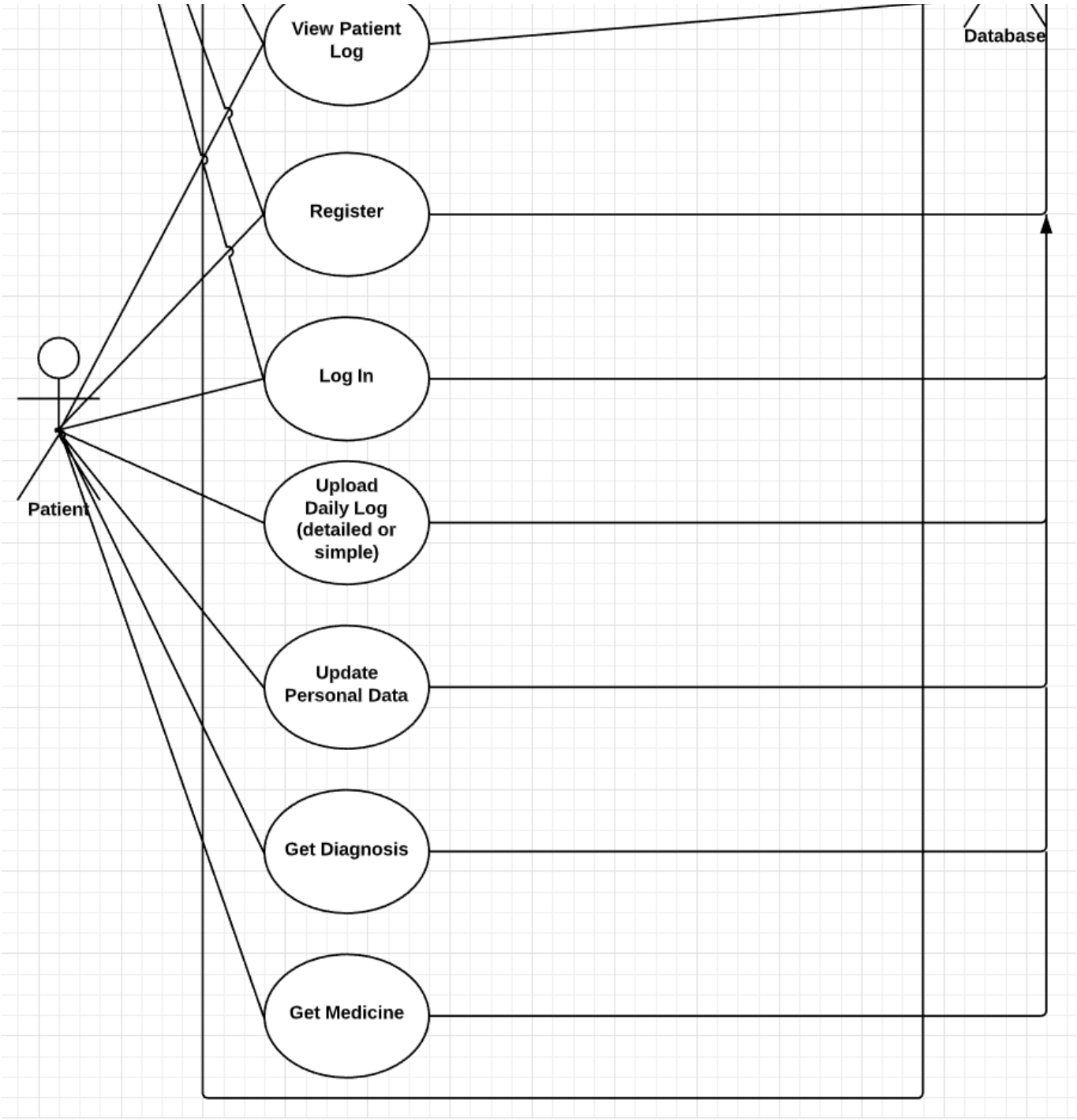
There are two main stakeholders in our project: **patients** and **doctors**. There are several **use cases** in our system regarding to two stakeholders separately:

- **Patient**: Patients want to add and view their daily logs easily, and they would wish to share this information with doctor. They also want the feedback from doctor which will assist them in obtaining more information and right medicine.
    - Patient adds simple/detailed daily logs.
    - Patient view their logs.
    - Patient view the diagnosis from doctors.
    - Patient view the medicine given by doctors.
    - Patient receives text messages reminding them to take medicine
    - Patient receives messages from their doctor.
- **Doctor**: Doctors want to access patients' logs and give their advice as well as the corresponding medicine. They also want to be able to communicate with patients without disclosing their personal phone number.
    - Doctor search patient by names.
    - Doctor views patient's logs.
    - Doctor gives the diagnosis to patient.
    - Doctor gives the medicine to patient.
    - Doctor send messages to patients.

Use Case Diagram

The figure below is the simple illustration of the use case situations, which contains the main functionalities of each.



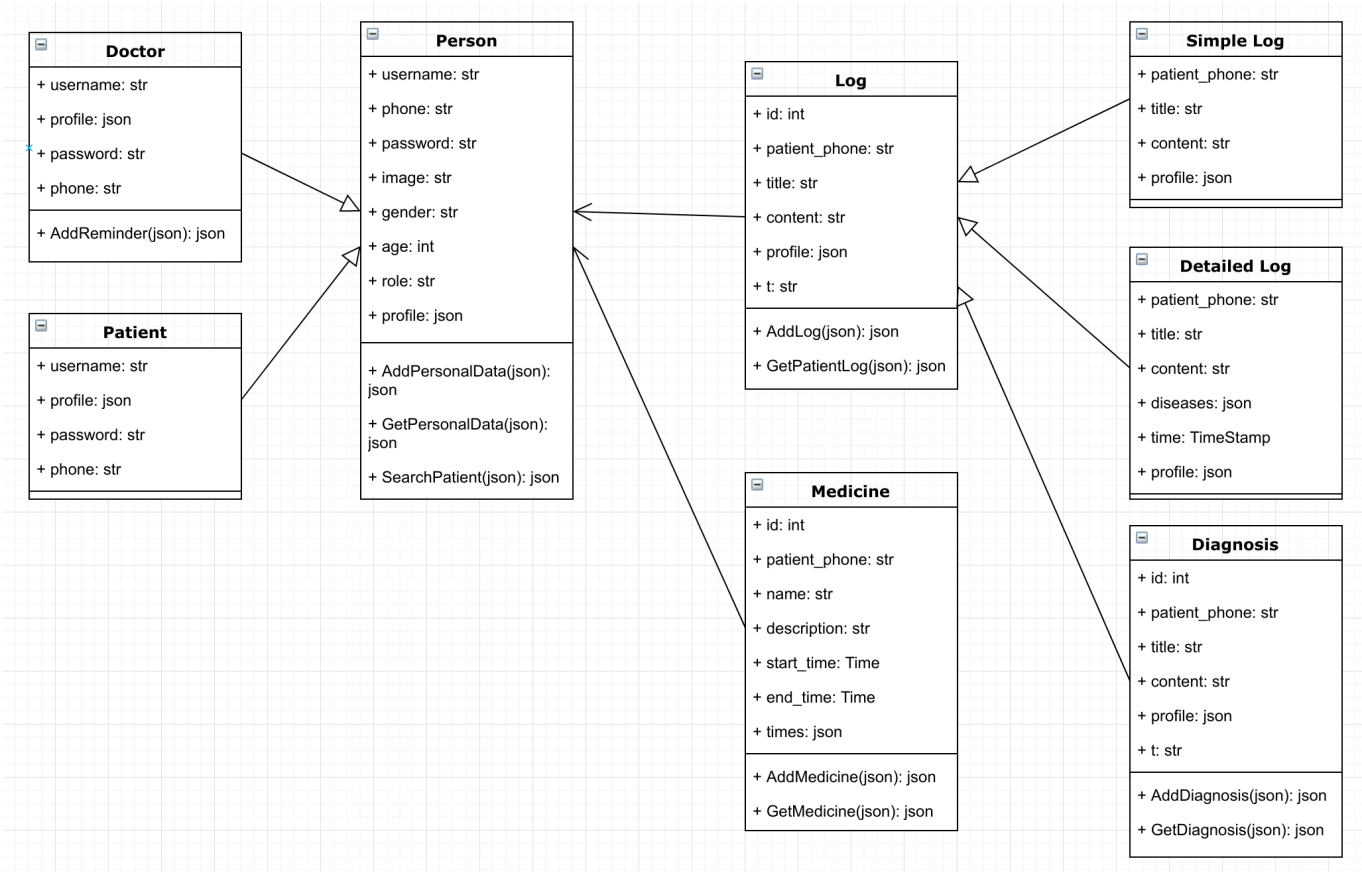


Architecture & Design

Class Diagrams

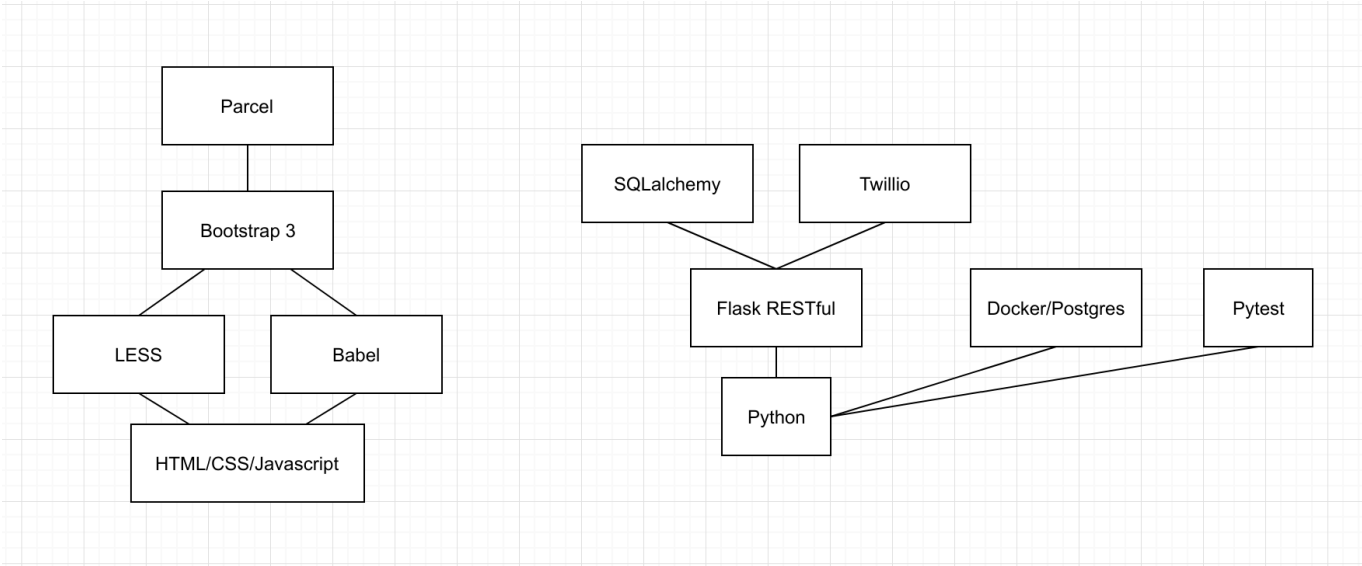
The figure below shows the class diagram of our backend. Our main class is the **Person** class and all the other classes are centered around Person and related to it. The frontend does not contain any classes and just serves

as a user interface for the API calls to the backend, thus we do not have a class diagram for the frontend.



Architectures

UML diagram of frontend and backend architectures



Frontend

Languages Used:

- Javascript
- HTML
- CSS

Tools Used:

- Babel (for legacy-Javascript support)
- Parcel (for packaging)
- Bootstrap 3
- LESS

We chose a development style that would maximize the amount of users able to utilize our website by ensuring near-perfect browser compatibility. **Babel** provides the ability to compile modern Javascript to legacy Javascript which allows us to use features such as `fetch` and Promises without worrying about compatibility issues.

Bootstrap 3 was used to provide a design framework which has great browser compatibility, as compared to **Bootstrap 4** which drops support for several browsers.

LESS allows us to simplify the development process of CSS styling but also allows us to compile CSS for legacy browsers.

All of our frontend source code is compiled, compressed, and packaged using **Parcel** a relatively new packager and compiler for the web. This simplifies our workflow while minimizing the network overhead for loading our website.

We test our frontend manually by clicking all the buttons on all pages. We make sure all the functionalities we have in our frontend works well including the components and the `Postman` API handlers.

Backend

Languages Used

- Python

Tools Used

- Flask RESTful
- SQLAlchemy
- Pytest (for unit test)
- Docker/Postgres (for database deployment)
- Twilio (for SMS reminder)

Our backend consist of several folders: `config` for handling configuration and authentications, `database` for connecting to the database set up by Docker, `docker` for setting up Docker, `error` for all the error handlers used in the system, `models` for all the database models/tables, `resources` for all the RESTful API handlers, `sms` for the utilities used for sending messages and `tests` for all the unit test scripts.

We use **SQLAlchemy** for interacting with our database. Because of this framework choice, we changed our backend to be written in an object-oriented style. In order to test all the features in our system, we choose **Pytest** as our framework since it can run all the test scripts in one single command. Our test scripts cover all the APIs except the one for adding reminders, which involves the SMS messages and we cannot test the feasibility of sending successful messages (for the failure may due to multiple reasons such as the Twillo server error or the mobile carrier's error). Additionally, in order to be able to spawn multiple instances of our database across the computers of all of our developers, we use **Docker**. This allows us to quickly start a development database on each developer's computer without interfering with any software currently running on their computer.

Reflections & Lessons Learned

Our Story

At the start of the development process, our team was planning on creating a security-centered EHR system for iOS. While initially this system seemed easily doable, as the idea was fleshed-out, it became clear that this was a huge undertaking, though still possible to complete in one semester.

As development processed, the team dynamic was less than ideal. While there were four people on the team, there was only 2 members who were actively doing work. Due to this imbalance in the team dynamic, it became clear that the goal of creating the target secure EHR system was not possible.

Therefore, our team changed our target application to create a more standard EHR system. However, the 2 people who were not contributing to the project continued to not contribute much. Therefore, our 4 people team split in half, creating 2 teams of 2. Due to the time restrictions imposed by changing our application half-way through the semester combined with the doubled workload of splitting into two teams, our team decided it would be best to constrain the objectives of our application.

Lessons Learned

- Big projects are only as possible as all the group members make it.
- Holding people accountable for their action or lack thereof is important but hard to do, especially when you are peers.
- The larger a development team is, the harder it is to manage well.