Pill Dispenser & Reminder

November 10th, 2019

Group 11

Mario Bocaletti

Henry Clay

Corey Nielsen

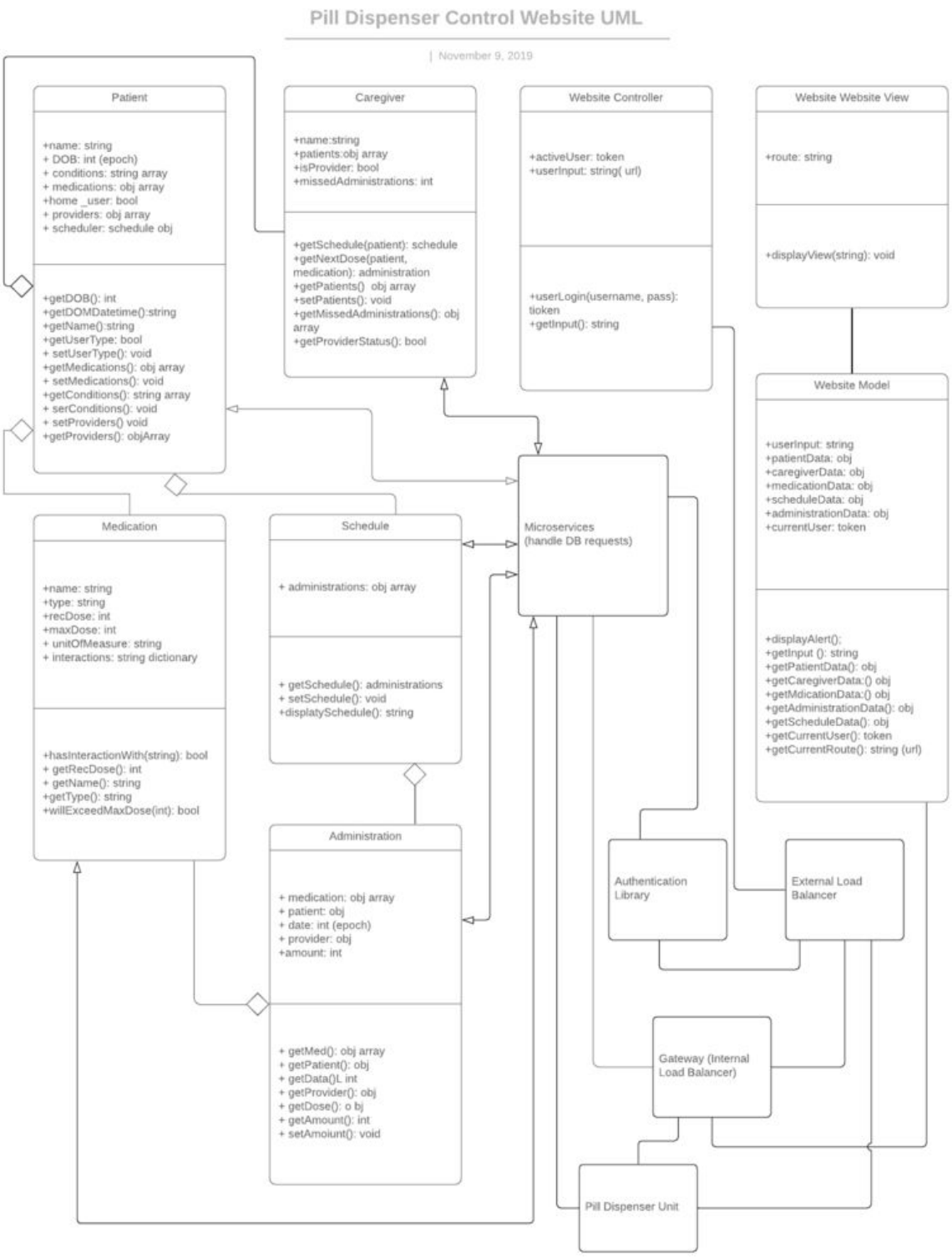
Samantha Tone

Pavan Thakkar

Customer

Emmanuel Rovirosa

UML Class Diagram



Packaging the Implementations

Because we're implementing microservices and microapplications, having loosely coupled units is implied, at least at the service/app layer. Microservices should have as few interdependencies as possible so that updating one microservice does not cause others to crash when one service is temporarily down. Applications should only serve content that is returned by the backend to REST API calls. The front end should load without regard for the state of other front end applications.

To take an example from our UML diagram above, suppose that the microservice responsible for interacting with the medication database and serving out details on medications (such as dosages and interactions) is temporarily down. The microservice responsible for maintaining the schedule and alerting users when medication is due should still alert users that a medication should be taken, and this alert should produce an alert sound and appear to the user in the front end. The scheduler service should give the dispense order to the pill dispenser when the user authenticates with the facial ID. The only symptom of the medications API being down should be medication details not showing up in the front end.

This loosely coupled approach would allow us to solve for certain services being down by implementing design patterns such as caching.

Design Assessment

Based on the requirement definitions (the system shall allow users to create accounts as patients, medical professionals, or family/caregivers, the system shall authenticate user credentials via face ID, etc) and the requirements specifications (a database exists to store application data such as patient schedules and medications, etc) that we wrote for assignment 2, our design would support iterative development well. The designed system's value is spread out over much of the system. When working on the implementation, it would make the most sense to get the whole system working relatively well and then add features throughout the system. We need to successfully implement the administration, authentication, website model, schedule, and medication portions of our design, and then go from there. There is some potential for code reuse that we would factor in when taking into consideration whether or not to develop iteratively or incrementally. It would not be as feasible to get part of the system working really well and then add more parts to the system (incremental development). Furthermore, it won't be necessary to complete the implementation of one part of the designed system before moving on to another.

Design Patterns

Since we elected to move forward with a microservices application, there are several useful design patterns that are available to such an architecture. There are many resources such as code generators (i.e., JHipster) and frameworks (i.e., Spring Boot, Django) that provide a strong starting point for any microservices project.

Proxying

We chose to use the proxying design pattern by means of a backend gateway in which the gateway acts as a single point of entry and proxy to several microservices. Our data flow from the data layer to the user is separated by the service layer as well as the gateway layer. This design pattern offers additional security to the system. The user is not able to reach the back end directly because the front end apps have to go through the proxy gateway, which validates authentication tokens.

Object Orientation

Our data model and authentication flow lends itself to an object oriented paradigm. We use object tokens to move authentication/user data from microservice to microservice; these objects include the scopes the user can access as well as other data such as user preferences. We also have entities that can be well represented as objects with relationships to each other that can be represented with pointers to other objects. For example, a patient can have a schedule object which in turn has medication objects that map to a date in a dictionary. Patients can also have arrays of medications representing their prescriptions. Caregiver objects can have arrays of patients that point to all the objects above. Object orientation would make this complex system.

Event Driven Design

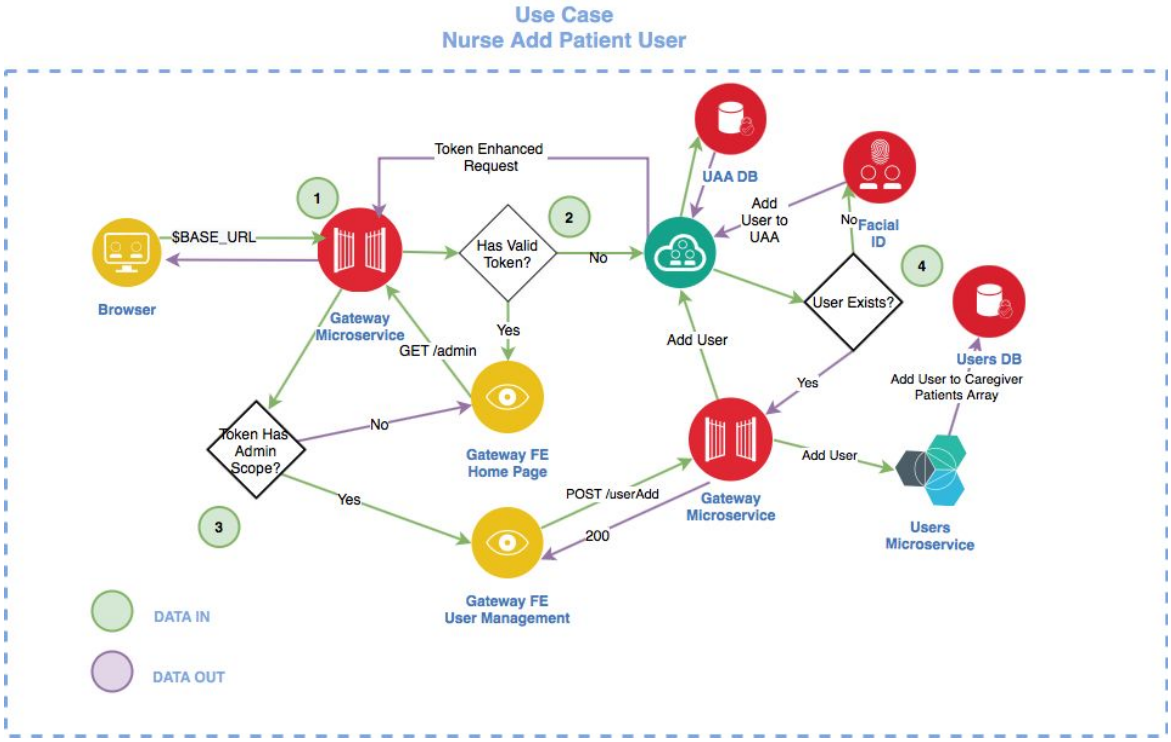
Our system's main functionality is to alert users when medication is due. This means that the workflow of the backend microservices must be event-driven. For example, the schedule should trigger an alert event; authentication after an alert should trigger a dispensation event; low supply of medications should trigger an alert.

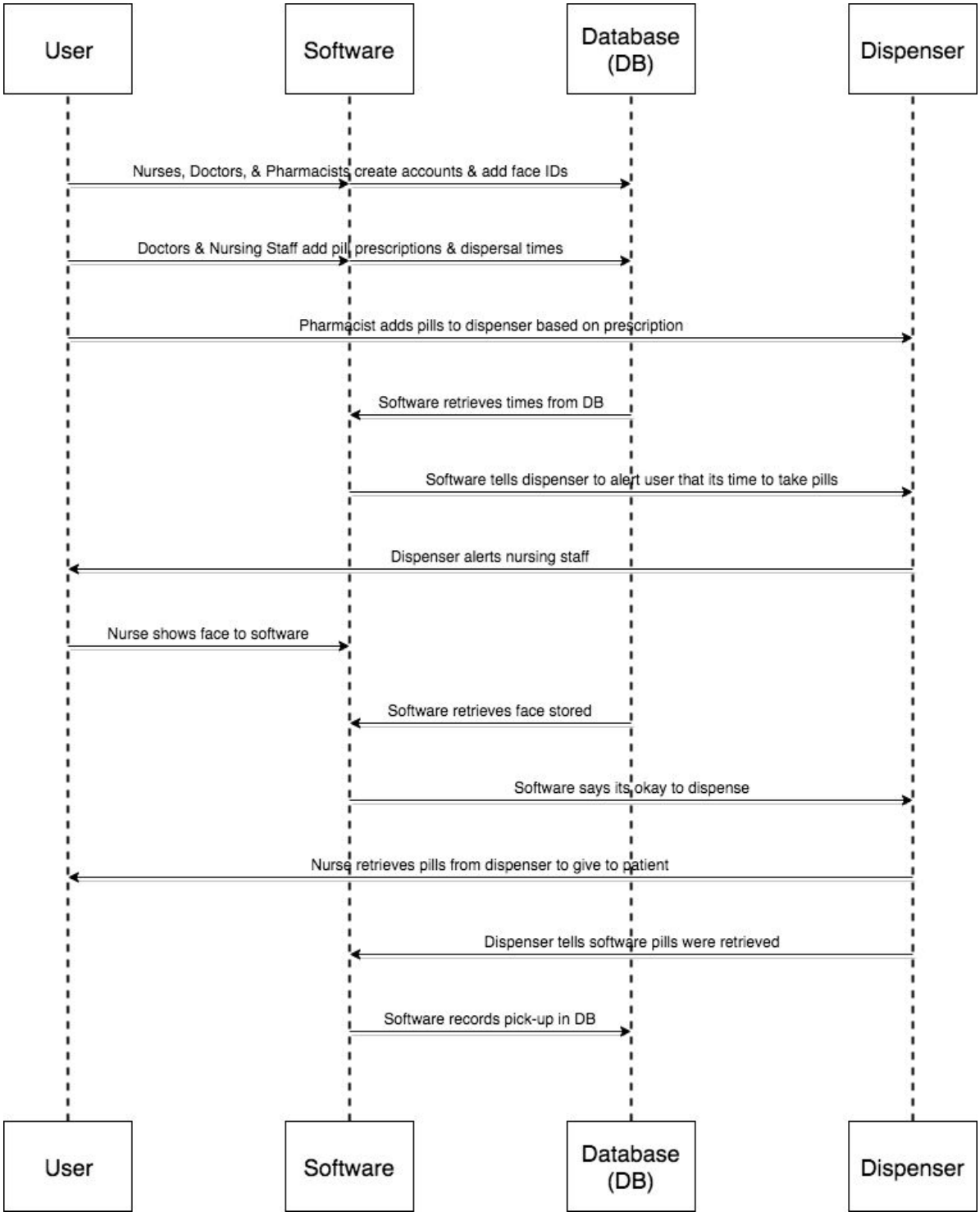
RSA Encryption/Decryption of Messages

Authentication is one of the most important part of our system because the system is designed to work in the healthcare industry. We can use standard RSA APIs to authenticate with tokens after initial authentication. The authentication flow would be to use a Facial ID API to authenticate the user with a UAA service. The UAA service would then generate an RSA encrypted token that would contain the authentication scopes of the user (i.e., what features the

user can access, the username, and any other simple data that microservices need to serve requests to particular users.

Use Case Sequence Diagrams




User
Software
Database (DB)
Dispenser
Nurses, Doctors, & Pharmacists create accounts & add face IDs
Doctors & Nursing Staff add pill prescriptions & dispersal times
Pharmacist adds pills to dispenser based on prescription
Software retrieves times from DB
Software tells dispenser to alert user that its time to take pills
Dispenser alerts nursing staff
Nurse shows face to software
Software retrieves face stored
Software says its okay to dispense
Nurse retrieves pills from dispenser to give to patient
Dispenser tells software pills were retrieved
Software records pick-up in DB
User
Software
Database (DB)
Dispenser

Interfaces

The defined interfaces are the hardware used for the pill dispensing, REST requests, device camera, & several inter-package interfaces. Detailed below are the interfaces we are working with as well as the inputs & outputs for each.

- **Pill Dispenser Hardware:**
    - Input: command to dispense, user authentication confirmation of pill collector;
    - Output: notification of pills being dispensed
- **REST Requests:**
    - Client Input: user login info (username & password), caregiver services call (permissions, outstanding notifications, assigned patients), patient services call (patient medical information, medication & dosage info), scheduling services (scheduled dispense times), facial recognition services call (image of user);
    - Client Output: normal REST responses;
    - Server Input: user login info accepted message (true or false based on successful authentication), generate medical report request, view patient medication information request, pill dispense request, update medication/dosage/scheduling requests;
    - Server Output: normal REST responses
- **Device Camera:**
    - Input: command to take photo;
    - Output: photo of user's face (hopefully)
- Inter-package Interfaces
    - **Front-end (GUI):**
        - Input: info from caregiver services, patient services, facial recognition services;
        - Output: user input information about caregiver (login info & general info) & patient (medication, dosage, medication, etc.) to caregiver & patient services
    - **Authentication:**
        - Input: username & password from GUI;
        - Output: a REST request to authentication server with username & password
    - **Caregiver Services:**
        - Input: user information & permissions details from GUI;
        - Output: user information & notifications for action on patient data to GUI
    - **Patient Services:**
        - Input: patient medical history & patient prescription data from GUI;
        - Output: patient medical history, notifications for unfilled data, notifications for delivery of pills to GUI
    - **Scheduling Services:**

        - Input: scheduling information from GUI;
        - Output: notifications of pill dispense times to GUI
    - **Facial Recognition Services:**
        - Input: image uploaded to GUI from device camera;
        - Output: confirmation of facial scan against the database of facial information (upload to facial recognition database if being used during sign-up) to be displayed on GUI & to send command to dispenser to unlock

Exceptions

Internal System Exceptions:

These are errors that occur while the program is attempting to carry out requests. These should be handled with ease by the error handling code in the software. Some errors that will be included in this will be mistakes in user input (invalid username or the date does not exist) or data that doesn't exist is trying to be accessed. These should all be handled by the error handling and not result in an extended disruption of service. In the case of bad input, the user would then be prompted to re-enter the information in the correct format.

External System Exceptions:

These can happen when requests made by the software to outside entities such as the dispenser fail. When the software looks to dispense a pill it will send a request to the dispenser to dispense it. If it fails, then it will keep trying for a set amount of times before an exception is called. This would notify the user that their dispenser is either physically broken or may not have any power. Any instance where the hardware(dispenser) fails to do the expected thing will be handled by the error handling and result in some notification being sent to the user pertaining to the exception that was caught.

Contribution Summary

The following indicates team member contributions during the project stages leading up to & including HW4:

- Mario Bocaletti - Completed the design patterns section, the use case sequence diagram, and the portfolio. Partnered with Sam in last minute completion of the Packaging the Implementations section.
- Henry Clay - Completed the UML diagram.
- Corey Nielsen - Completed the Exceptions page.
- Samantha Tone - Completed the Interfaces & compiled all of the sections from the group into the HW4 final document. Partnered with Mario in last minute completion of the Packaging the Implementations section.
- Pavan Thakkar - Completed the assessing of how well our design would support incremental or iterative development, based on the requirements and the potential for code reuse.