HealthHub Final Documentation

Roberto Noel – Alex Bogdanowicz – Ricardo Chacón – Jack Schulz

December 19, 2019

GITHUB REPO: /ElChackAttack/EHR

Contents

1 Introduction

Health Hub is a lightweight electronic healthcare records system (EHR) designed from inception with the focus on increasing Patient-Clinician interaction while also decreasing logistical overhead for all users. As of 2017 85.9% [1] of all hospitals within the United States have implemented some form of an EHR system, yet many of these systems fall short. All EHR systems at their core inherently partially achieve this goal, but it is in their implementation that falls short. An example of this failure comes from the Director of the Nudge Unit at the University of Pennsylvania, Mitesh Patel, who found 85% [2] of post-cardiac arrest patients were not receiving vital follow up care because of redundant paperwork physicians were forced to complete. Physicians were required re-enter patients' basic information, that already existed within their EHR database, while attending to their existing duties. There simply isn't enough time in a physician day to provide the best care for their patients while wasting time refilling out paperwork. Automation was the solution for Dr Patel's cardiac crisis, and automation is one of two pillars of our EHR system. The second: breaking down the barriers between Patient-Clinician communication.

2 Requirements & Specifications

2.1 Stakeholders

Stakeholders represent the individuals for whom the EHR system is being developed, and can include users, customers, maintainers, investors and more. We provide a breakdown of the expected stakeholders as follows:

1. Users: these are the individuals who will be interacting with and benefiting from the features implemented in the EHR System
   (a) Patients: users with the capacity to book appointments, view schedules, medical history, prescriptions, get recommendations, check symptoms, look up hospital staff, etc.
   (b) Physicians: users that may move appointments, check medical histories of all direct patients, issue prescriptions, offer recommendations, look up other hospital staff/patients etc.
   (c) Nurses: users that may see schedules, input data, and look up hospital staff/patients etc.
2. Maintainers:
   (a) Technical Administrators: able to maintain stability and make corrections where necessary, able to view changes in the system to track changes in a well documented manor etc.
3. Customers:
   (a) Hospitals: represents the target customer for our product. Hospitals needs, practices, and existing infrastructure will have to be taken into account throughout the development of the EHR System

2.2 Core Requirements

1. User login interface: this user is defined as any person who through an electronic device logs in to the system. This definition allows for specific roles for the type of user currently accessing the platform. For instance: primary physicians have higher security clearance.
2. Medical History: View most recent checkup and health information at a glance in a brief and concise format. View recent, and past test results on an additional tap within the patient information page. Detailed patient records can also be found under an additional separate tab.
3. File Management System: Encrypted file storage for sensitive information files (e.g. Allergy Records, Blood Tests, etc.)
4. Transaction Log: IT administrators have a tool to see changes in the system. The transaction log will resemble the insertion/deletions log infrastructure commonly seen in Git logs.

2.3 Major Functionalities

1. Appointment and scheduling system: patients are able to request check-ups with their primary care physician.
2. Q&A Social Forum: an anonymized forum with features such as support groups, Q&A, and certified personnel feedback/responses. This feature also includes user-to-user chat functionality.
3. Automated Reminders: Email reminders for taking medication and refilling prescriptions

2.4 Feature Specifications

1. Medical History
   - Physicians search for patients by last name, and then enter the new checkup information.
   - Able to see past medical history at a glance to allow for easier recognition of change in a patient's basic checkup.
   - Patients are able to see their own checkup information, without the long process of requesting the files from the hospital.
2. File Storage
   - Medical Staff upload and view patient files.
   - Medical Staff have easy access to lab tests, medical reports etc, for each patient
   - Patients are able to view their own medical files, without the difficult bureaucracy typically associated with requesting medical records.
   - Patients are not able to upload their own files to maintain site security.
3. Transaction Log
   - Integrated through the Alembic system.
   - High portability thanks to the ability to run templated queries through Python scripts. This means that further development for the platform is convenient and easy for anyone who reads this documentation.
   - High security which stems from the unique integration patterns of permissions for our system.
   - Difficult to copy our unique integration of the EHR system, making it consistent with the stakeholders' requirements. This is ultimately the vision and principle under which the group develop the platform and it is with great satisfaction that the group is able to present this final deliverable.
4. Appointment Scheduler
   - User can schedule appointments for themselves only.
   - Medical staff can add schedules for any patient through the patient search.
   - Upon creation of a schedule through any user a new row is populated and an eventually notification relating to the appointment is queued.
5. Social Forum
   - Users can write forum posts and comment on said posts.
   - Users are able to subscribe to sub forums that they can relate to.

- Users can see the top posts of the last day.

6. Automated Reminders
   - In regards to prescription consumption our system offers users the option to receive notifications daily to remind them to take their medications
   - When a patient has an upcoming appointment our system pushes an email notification to remind them of their upcoming appointment.

3 Process

3.1 Chosen Methodology

Our team used Spiral Model for iterative development. This model, first proposed by Barry Boehm PhD. in 1986, allows for continuous development on a large scale platform while balancing security and risk mitigation components [3].

The high-criticality of an EHR application initially led us to consider strict, plan-driven methodologies such as the Waterfall method. Our development, however, was constrained to a four month time-frame, and it would be difficult to implement all of the required features with this kind of methodology.

The Spiral Model allowed us to strike a balance between traditional plan-driven approaches and strictly iterative strategies. By doing this we were able to avoid getting tied down by planning while at the same time prioritizing security.

3.2 The Spiral Model

The structure and characteristics of the Spiral Model are laid out in a July 2000 report from Boehm's Spiral Development Workshop titled *Spiral Development: Experience, Principles, and Refinements*. Key features mentioned are:

1. Concurrent rather than sequential determination of artifacts.
2. Considerations for each spiral:
   (a) Critical-stakeholder objectives and constraints
   (b) Product and process alternatives
   (c) Risk identification and resolution
   (d) Stakeholder review
   (e) Commitment to proceed
3. Uses risk considerations to determine the level of effort to be devoted to each activity within each spiral cycle.
4. Uses risk considerations to determine the degree of detail of each artifact produced in each spiral cycle.
5. Manages stakeholder life-cycle commitments with three anchor point milestones:
   (a) Life Cycle Objectives (LCO)
   (b) Life Cycle Architecture (LCA)
   (c) Initial Operational Capability (IOC)
   (d) Stakeholder review

   (e) Commitment to proceed

6. The model has four phases. A software project repeatedly passes through these phases in iterations called Spirals:

   (a) Planning: Requirements are studied and gathered in this phase. It includes estimating the cost, schedule and resources for the spirals. As the product matures, identification of system requirements and unit requirements are all done in this phase.

   (b) Risk Analysis: Risk Analysis includes identifying, estimating, and monitoring technical feasibility and management risks, such as schedule slippage and cost overrun. Once the risks are identified, risk mitigation strategy is planned and finalized.

   (c) Engineering: This phase refers to production of the actual software product at every spiral. Actual development and testing of the software takes place in this phase. It includes testing, coding and deploying software at the customer site. A POC (Proof of Concept) is developed in this phase to get customer feedback.

   (d) Evaluation: After testing the build, at the first phase, the customer evaluates the product and provides feedback.

3.3 Spirals

We broke our development timeline into four spirals.

1. First Spiral
   - Most of our time was spent planning out the tools we would use and laying out a foundation upon which we could rapidly iterate. This included creating our database, and creating base-versions of our flask components.
   - We also spent time creating stub versions of our features that we would later go on to develop further with each following iteration.
   - Lastly, we fully developed registration and login authentication as well as a logging system to keep track of changes to the database.
   - Upon evaluating our system, we found that our framework would make the development of our software efficient, as most tasks were independent of each other and could be developed in parallel.
2. Second Spiral
   - With a strong foundation built, we began developing front-end versions of our stubs. This made conceptualization for the back-end easier and provided smoother communication for pair programming.
   - We also established a better verification system with confirmation emails upon registration.
   - Nearing the end of this iteration, we spent some time dividing responsibilities and establishing clearer roles within the team. This proved vital to completing the next spiral more efficiently.
   - Upon evaluation we saw that our visualization of each feature increased the tangibility of our program, and we were ready to develop our backend more thoroughly.
3. Third Spiral

- In this spiral, we began the rapid development of each core requirement as well as major functionalities. We used pair programming to implement the back-end and front-end in parallel.
- We also introduced new tools such as Redis and Celery into our system to perform asynchronous tasks such as updating the top posts on the social forum and sending out email notifications.
- Lastly, we designed our user flow and developed a more visually appealing front-end to provide a pleasant user experience.
- During the evaluation phase of this spiral, we reflected on the progress that we had made and realized that although we technically had working versions of all our features, we there were sub-functionalities that we hadn't initially thought of.

4. Fourth Spiral

   - In our last spiral, we spent time polishing our features. This included refactoring our code and making sure our system passed white and black box testing.
   - We also added necessary sub-functionalities such as the ability for users to search for forums, edit their bios and leave comments.
   - Lastly, we made sure each type of user had specific access to their corresponding functionalities. We used python decorators to ensure that users were restricted to their specific use cases.
   - While evaluating this iteration, saw room for improvement in the security of our file management system. URLs for the stored files should be hashed to prevent unauthorized access. This is something that will be further developed in future iterations.

4 Software Architecture Structure

4.1 Introduction

The core of our EHR system is rooted in a popular micro web application framework, Flask, written in Python. Compared to other Python application frameworks, Flask proves particularly advantageous given its independence from any specific libraries or tools [4], functioning primarily as a framework and giving freedom of choice from modules to routing engines back to the developer. As is popular in most web-development architectures, the HealthHub application is based heavily on the Model View Controller design pattern with Flask primarily playing the role of the controller. The application utilizes a traditional MAMP MySQL Database as the Model which integrates with the MVC framework via Flask-SQLAlchemy [5], which is an Object Relational Mapper that, in addition to providing ORM security features (e.g. combatting SQL injections), features particularly great integration and session management via the Flask controller. Finally, View's are handled using Flask's native blueprinting system - on the controller side - and the Jinja2 templating engine, which provides for a more "pythonic" front end, creating for smoother back-end front-end interactions, given that Python is not natively suited for views and interfaces.

4.2 Model

Whereas Model definition is an ongoing process as the application scales, grows in features, and provides new functionality, instantiating an E-R diagram is one of the initial aspects of a requirements analysis phases. This proves to be especially true in an MVC framework, especially given SQL-Alchemy's tight integration with Flask's Login Library [6], which integrates with SQL-Alchemy defined models to pass a user instance along with a user session, as an efficient and effective way to pass user contexts across views. The system also extends to permissions.

Figure 3. E-R Diagram, details the Entity Relationship schematic that closely corresponds to our model definitions. Conceptually, the database model can be grouped into Users & Attributes, Forum, Task Queues, and Scheduling, which closely correspond to our major application functionalities.

The User table definition is particularly noteworthy as instances of the User class - wrapped in the current_user instance - thanks to Flask's Login Library, are passed around with the application context and are an integral component of Flask's functionality. This capacity to function both as the database model and as an instance of the User class, allows the model to perform core functionality, from permission handling and storage, (i.e. through roles which are defined and checked using bitwise operators), to queuing user specific asynchronous tasks (e.g. sending confirmation emails or updating specific tables), to hashing passwords upon registration using the werkzeug security library [7]. Permission handling is made easier by the implementation of a decorator which call upon the user instance to check whether a user has a certain permission, or whether they have a specific role.

```
def permission_required(permission):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if current_user.is_administrator():
                return f(*args, **kwargs)
            if (not current_user.can(permission)):
                abort(403)
            return f(*args, **kwargs)
        return decorated_function
    return decorator
```

Sample Code 1. Permission Decorator

4.3 View

In addition to Flask's integration with Jinja2's templating engine, an important consideration to the functioning of views in the HealthHub application are Flask Blueprints. Blueprints are Flask's way of factoring the application into function-specific packages, in the case of the HealthHub application, each having access to their own views and forms [8]. At the creation of an application context, blueprints are registered with the flask app, consisting of URL prefixes that specify routes/domains, further integrating the MVC framework. The HealthHub application consists of the following major blueprints:

- Auth : handles logging in and registering
- Forum : handles Forum related tasks (e.g. posting, profiles etc.)
- Health Check : handles Checkup related tasks
- Main : handles the landing page & initial routing
- PDF Viewer : handles viewing of documents
- Prescript : handles prescription management
- Profile : handles patient profile views and search
- Sched : handles scheduling of appointments
- Upload : handles uploading of files

As is evident from the naming convention, each blueprint functions as a separate "package" of functionality, which are also mapped to different aspects of the application, thus ensuring high levels of cohesion and low levels of coupling across the HealthHub application structure.

4.4 Controller

As previously mentioned, Flask's main function, apart from functionality generated via the integration with external packages, is that of the controller in the MVC framework. Flask stitches together the routing mentioned in the section on Views via it's blueprint engine and integrates it further with the database Model (e.g. SQL-Alchemy instances). As Flask is an instance based framework, each instance features its own configurations (e.g. development configurations etc.) which allows the application to feature multiple instances with different configurations, as is done for Health Hub's confirmation email functionality or redis/celery based asynchronous tasks (see Asynchronous Task Management).

From the perspective of the views, controllers make up the views.py files of each Flask Blueprint. For example, in the Forum blueprint, the view decorated with:

@forum.route('/profile', methods = ["GET", "POST"])

Which renders the template:

return render_template('forum/profile.html', posted_posts = posted_posts, [...] forum_pro = forum_pro)

is effectively the controller, the view being the rendered html file "forum/profile.html". The model is passed to the views through Jinja2, bringing together a simple but accurate example of the MVC framework governing the HealthHub Software Architecture.

4.5 Testing

In any realistic development setting, testing not only factors as a large portion of time spent over the development process, but also the budget. The HealthHub application features a suite of automated tests, both white and black-box, which cover a decent portion of the application's functionality. Whereas an agile methodology the likes of extreme programming demands significant continuous testing, especially in regards to regression tests, a more plan-driven methodology (or hybrid as we have adopted) can be more lenient.

In any case, Flask features a comprehensive set of test-suites from which to implement unit tests, and we take advantage of these, in the form of the unittest module [9] in addition to the classic black-box testing tools offered by selenium. As the user model is core to the functioning of the application as a whole, we pay particular attention to creating a comprehensive suite to test user registrations, logins, roles and permissions, as well as other user functionality.

Our selenium test cases cover major application use cases, such as posting to a forum, scheduling an appointment, searching for a patient, or assigning new prescriptions. This ensures that, via forms or clicks, we have automated regression tests that are run after each new iteration and ensure that our application is working as it was prior to changes. Coverage reports are available via the coverage module [10] and all unit-tests and black-box selenium test files beginning with "test" are run automatically via the shell application context.

4.6 Front End

1. HTML/CSS

- Our front-end is built primarily in HTML and CSS, using Javascript only when strictly necessary. We use Jinjia to pass data from our queries to the front end. In many cases, this helps us avoid having to use Javascript since our team has limited experience with this language.

2. Flask-Bootstrap

   - Flask-Bootstrap helps us avoid copying code by creating inheritable blocks which can be passed to new pages using Jinjia. As we move down the user-flow, sub-pages extend parent or "base" templates. These include common widgets such as the side navigation bar present in every page of our website (with the exception of login and registration). This adds to the modularity of our program, reducing tight-coupling of our front end. When we want to edit the navigation bar for example, we don't have to go through each sub-page and paste the new code, instead the base is automatically inherited.

3. WTForms

   - Flask also provides us with WTForms, a package which allows us to quickly integrate the back-end structure of our forms with the front-end. We use Jinjia to instantiate WTForms objects, which automatically generate fields for user input. The package also allows us to stylize each field with our custom classes.

4. CSS Grid

   - To lay out each page we use CSS Grid, which allows us to create grid containers for our HTML objects. This helps us specify the location of each item on the screen and properly use our screen real estate. It also provides scalability to our users, with each page adapting to the screen size and dimensions of their device.

5. JavaScript

   - The primary function of JavaScript in our front end is for the search bar. We designed an auto-complete search bar which removes the need for a search result page. The program begins searching once two or more characters are typed and the user simply selects an option from the drop-down menu.

4.7 Factoring and Refactoring

Factoring is a core concept related to Flask's blueprint engine, and allows us to ensure that, at least on the level of modules or views, there is very little coupling, and large amounts of cohesion. Thus refactoring is only particularly necessary in the scope of a module, which we had periodically done, as in the case of the Forum, where specific, repeatable calls for subscription instantiation or post creation were modularized as functions and callable within the context of the blueprint. This helped to strengthen the cohesion of our Forum blueprint while mitigating function cyclomatic complexity. However, as can be noticed from a second look at the Forum blueprint, certain controllers/views will require refactoring in the future as a specific issue with the Flask.

5 Application Design: Component Breakdown

5.1 Forum View Controller Interaction

In order for the functionality of the forum to make sense in the context of a UML-diagram, certain conceptual liberties were taken. As seen in Figure 1. Forum View Controller UML, Home appears as a single class while in reality it is a series of separate view controllers rendered across multiple templates. The autocomplete box at the top sends a get request to the database on a per character basis following the “@autocomplete” route. Upon a search selection a new template is rendered through the forum-page view. At each stage in the uml diagram when a method is called a post request is sent to database followed by current template reloading and updating (in the case of liking, commenting, and updating the bio) or a new template is rendered, like in the case of the creation of new forums. A similar UML-diagram based on the same assumptions is available for the Searchable Patient Profile, and can be viewed in Figure 2. Patient Profile UML.

5.2 Asynchronous Task Management

A core aspect of any application is the capacity to handle asynchronous tasks. An application shouldn’t hang for a User because an email is being generated, file is being downloaded, or, in the context of regular use, a table query is being executed each time a user accesses a view, rather than on a regular update schedule. The HealthHub application thus integrates redis[11] as a message broker along with rqworkers for asynchronous task execution as well as celery[12] for asynchronous task-scheduling, specifically in the context of user notifications.

An integral component of HealthHub’s user functionality comes from the application’s capacity to send reminders to users on the basis of their prescription schedules. A celery-beat schedule is initiated in the config file detailing the name of the celery task function and it’s schedule (e.g. every 10 minutes), which allows the application to asynchronously run background tasks such as sending email notifications to remind patients to take their prescriptions. A sample task is illustrated below:

```
@celery.task(name = "demo_task_name")
def queue_reminders():
    if datetime.now().hour > 7 and datetime.now().hour < 23: #
        datetime.now().hour > 7 and
        notify = Prescription.query.filter((Prescription.notify ==
            True) & (Prescription.active == True)).all()
        for prescript in notify:
            if prescript.last_notified  < datetime.now() -
                relativedelta(hours = prescript.time):
                user = User.query.filter(User.user_id == prescript
                    .patient_id).first()
                current_app.task_queue.enqueue('app.tasks.' +
                    "send_reminders", None, prescript, user)
```

Sample Code 2. Celery Task Demo

6 Reflections and Lessons Learned

Throughout this project, we got a better idea of the skills necessary to perform well in a professional enviornment. A key takeaway is the importance of proper time management and division of work-load. Towards the end of our project we became better at managing our schedules. Hosting meetings to communicate the availability of each team mate provided the predictability needed to develop our program within our intended time-frame. We also focused on capitalizing on the skills of each team member. Assigning the right roles to the right people allowed us to complete each task much more efficiently.

We also learned how each component of a full-stack program is connected to provide the end product. While each person specialized towards a back-end or front-end role, pair programming made exposed each member to several interconnected aspects of Flask, a framework that was new to all of us.

Ultimately this project helped us learn the difference between a software engineer and a coder. Every member of our group has been a coder for several years, but for some of us, this was our first experience developing a full-stack program.

A good coder implements algorithms efficiently, writes legible code, and understands the syntax of several programming languages. A good software engineer on the other hand, knows how to plan the development of a program, find the right resources for each task, communicate effectively with his or her team mates, and collaborate with a broader vision of how the code provides value to its users.

7 Figures

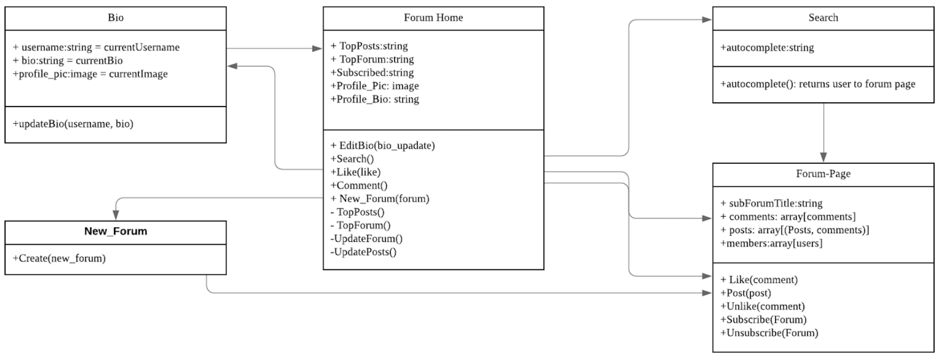

Figure 1. Forum View Controller UML

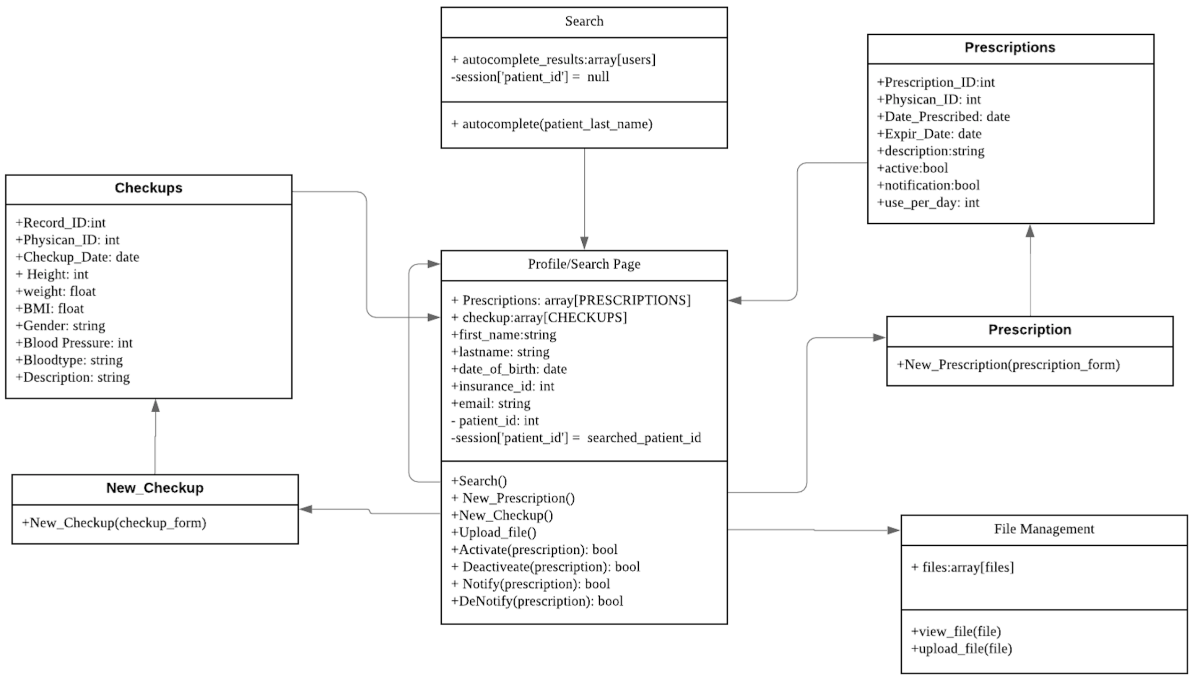

Figure 2. Patient Profile UML

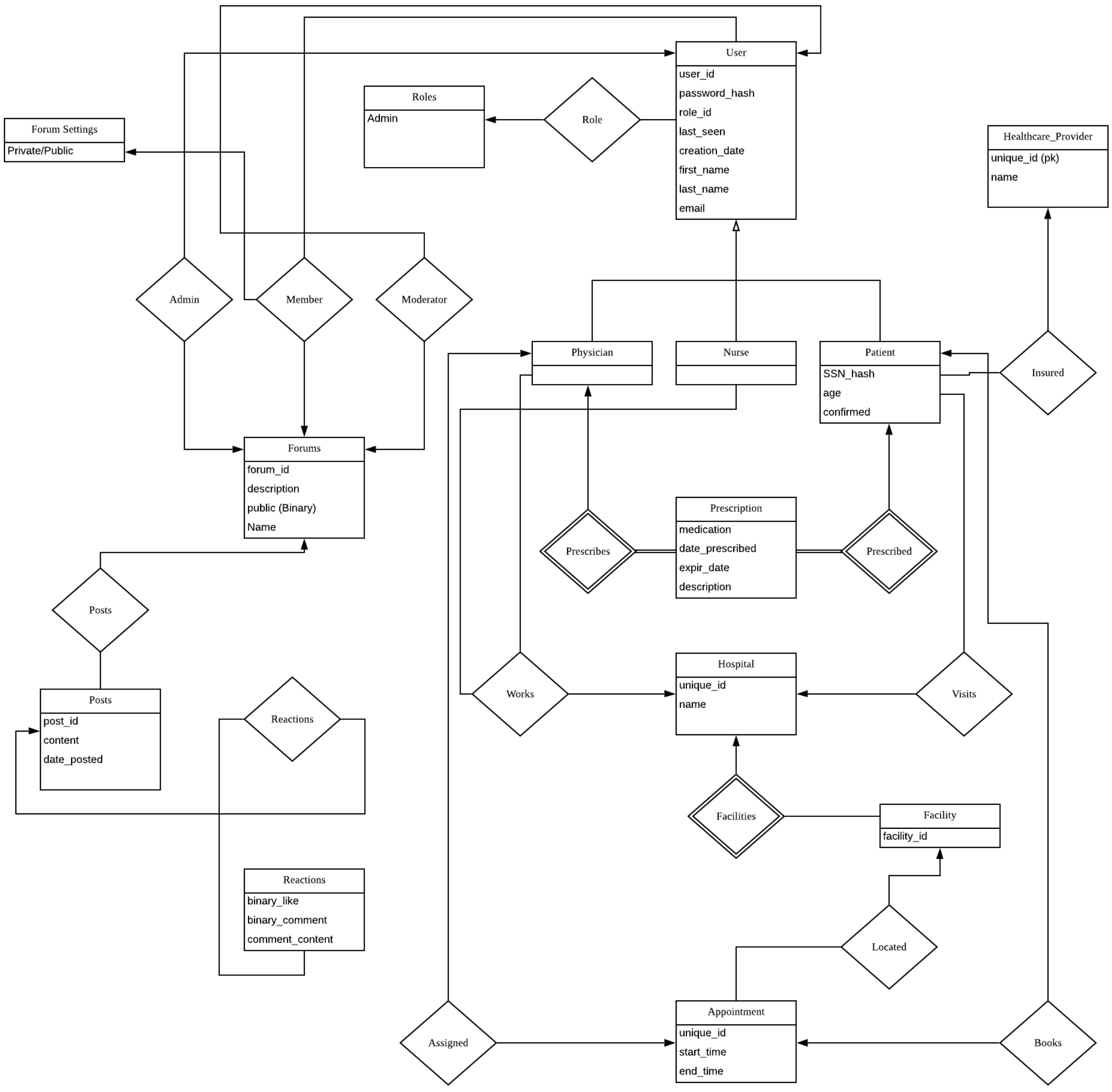

Figure 3. E-R Diagram

References

[1] https://dashboard.healthit.gov/quickstats/quickstats.php

[2] https://nudgeunit.upenn.edu/projects/using-default-options-increase-referral-cardiac-rehabilitation

[3] https://www.toolsqa.com/software-testing/sdlc-spiral-model/

[4] http://flask.palletsprojects.com/en/1.1.x/

[5] https://flask-sqlalchemy.palletsprojects.com/en/2.x/

[6] https://flask-login.readthedocs.io/en/latest/

[7] https://flask.palletsprojects.com/en/1.0.x/logging/

[8] https://flask.palletsprojects.com/en/1.0.x/blueprints/

[9] https://flask-testing.readthedocs.io/en/latest/

[10] https://coverage.readthedocs.io/en/coverage-5.0/

[11] https://redis.io/

[12] http://celeryproject.org