



MedLife

Design Document

GROUP 8:

Bipul Kumar (2014CS50282)

Kartar Singh (2014CS50286)

Prachi Singh (2014CS50289)

Gulshan Jangid (2014CS50285)



Table of Contents

1. Introduction:	3
1.1. Purpose:	3
1.2. Scope:	3
1.3. Overview:	4
1.4. References:	4
1.5. Definitions and Acronyms:	4
2. System Overview:	5
2.1. Architectural Design:	5
2.1.1. Block Diagram:	5
2.1.2. Class Diagram:	6
2.2. Module Definitions:	7
2.2.1. Notifications:	7
2.2.2. Search:	7
2.2.3. Patient timeline:	7
2.2.4. Messages:	7
2.2.5. Profile:	7
2.2.6. Authentication:	7
2.2.7. Access list:	7
2.2.8. User:	7
2.3. Technology/ Tools Used:	7
2.3.1. Client-side / Front-End:	7
2.3.2. Server-side / Back-End:	7
2.3.3. Database:	7
3. Detailed design:	8
3.1. Module APIs:	8
3.1.1. Registration and Authentication:	8
3.1.2. General User Information:	8

3.1.3. Patient:	9
3.1.4. Message:	9
3.1.5. Lab:	10
3.1.6. Doctor:	10
3.2. Database Design:	11
3.2.1. Data characteristics:	11
3.2.2. Database Functionality - CRUD:	11
3.2.3. Sub-databases:	12
3.3. Screen Layouts:	12
3.4. Use Cases:	17
3.4.1. Sequence Diagrams and High Level Code:	18
4. Deployment Design:	27
4.1 Environment:	27
4.2 Version Control:	27
4.3. Testing and Debugging:	27



1. Introduction:

In daily based scenario, if a doctor recommends a patient to get some required tests done for a diagnosis, they first need to wait at the pathology labs for their turn for the test, then they have to wait to take their test reports from the labs and again make an appointment at the doctor's for the diagnosis. The time taken for this process usually takes days as he has to wait for appointment at every step and the process entirely resides on the patient (or his family member) being the mediator of reports from one end to the other. Also, the absence of having a common platform for the patients and doctors to maintain the medical history and prescriptions of each patient is a cause of inconvenience. In this document we intend to tackle these problems, and propose a seamless solution.

1.1. Purpose:

The purpose of this document is to detail the requirements of "Patient Report Application", a seamless framework that allows users to have an organised system to maintain the medical history of the patient. This platform thus also serves as a system where patient can directly get his reports from labs and comments on his reports from his doctor without making several visits and appointments. It will always reflect the current state of the system architecture, and every deliverable will be based on the description as given in the document. It is intended to be updated and revised as necessary, to reflect the iterative nature of design and the continual development of the system.

1.2. Scope:

"Patient Report Application" is intended to be a cloud based and scalable free-to-use application that will be -

1. An easy to use framework, that allows the patient to maintain his entire medical history
2. A common platform between Labs, Patients and Doctors to facilitate the exchange of test reports.
3. A messaging system between Doctors and Patients so that patient can use this portal ask relevant doubts to his doctor.

Main aim is to coordinate the interaction between Pathology labs, doctors and patients. There is a host of features planned to streamline and improve user experience, including messaging, real-time notifications, and segregated profiles.

1.3. Overview:

The remainder of the document is organised into three sections -

- System Overview: an overview of the system functionality, interfaces and interactions
- Detailed Design: the design of each of the components in detail - the module design, system APIs, database design, screen layouts as well as use cases
- Deployment Design: deployment architecture and implementation specifications

1.4. References:

- [Unified University Inventory System](#)
- Design Format provided on moodle

1.5. Definitions and Acronyms:

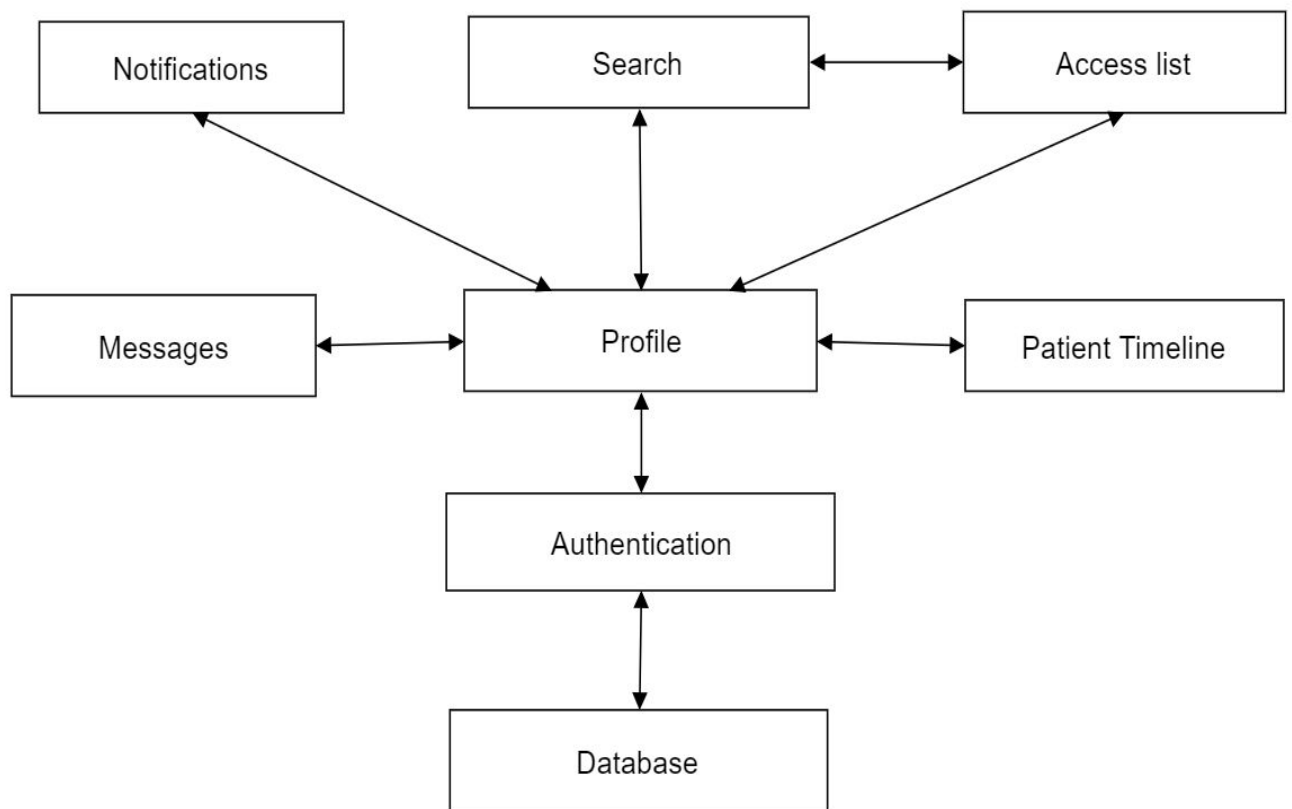
Term	Definition
User	Someone who interacts with the system: Patient, Lab or Doctor
API	Application Programming Interface
UC	Use Case
GUI	Graphic User Interface
DAL	Data Access Layer
UML	Unified Modeling Language



2. System Overview:

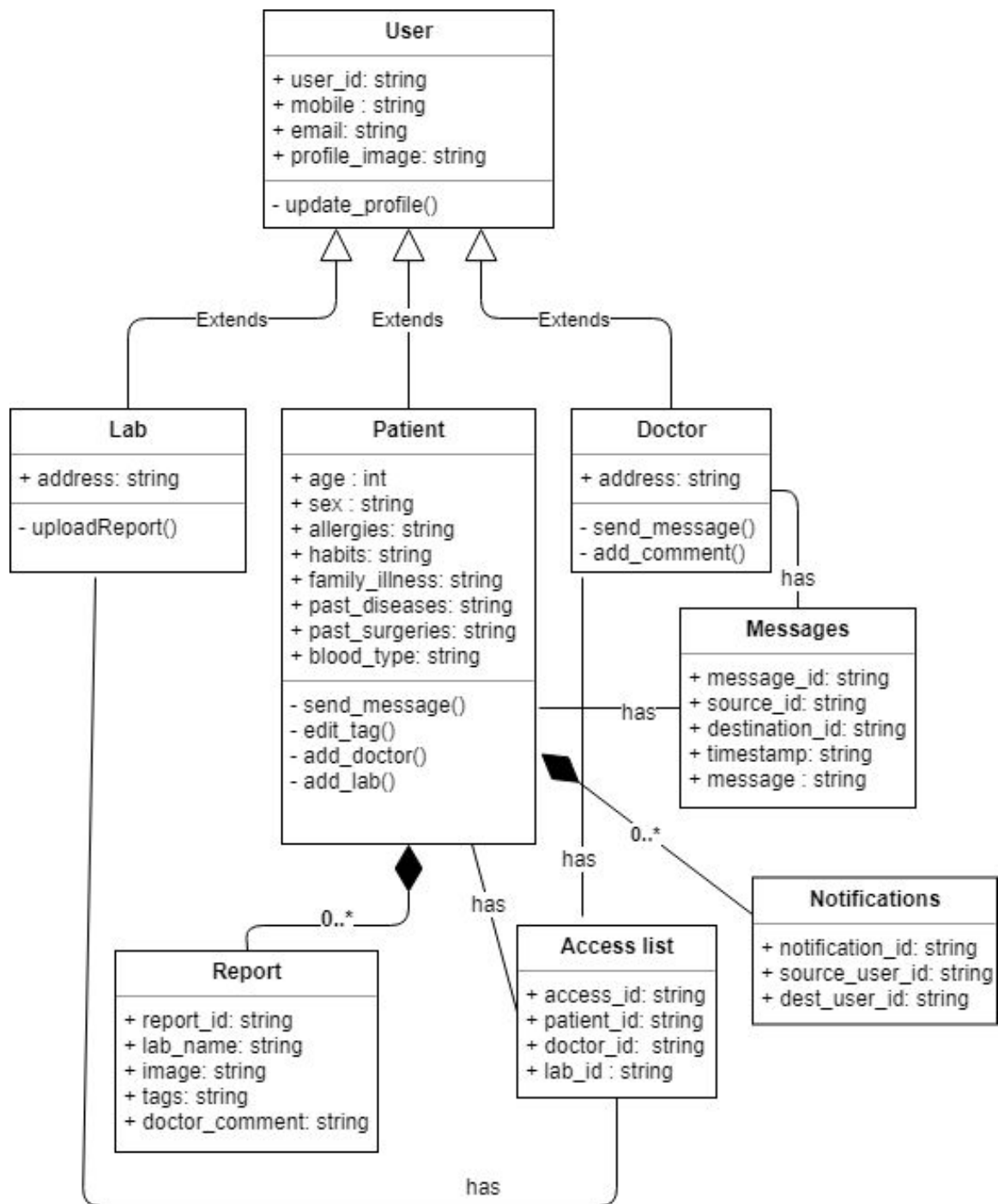
2.1. Architectural Design:

2.1.1. Block Diagram:



Various interacting subsystems

2.1.2. Class Diagram:



Classes comprising the modules

2.2. Module Definitions:

2.2.1. Notifications:

Used to notify Patient whenever lab uploads a report for that patient

2.2.2. Search:

Module to search database given keys. Different type of searches include patient searching for a doctor or patient searching for a report tag within his timeline.

2.2.3. Patient timeline:

Module which corresponds to a particular patient and contains the reports that have been uploaded to that patient. Patient can edit tags within a report in his timeline

2.2.4. Messages:

This module handles the messages sent by patient to doctor or doctor to patient.

2.2.5. Profile:

Corresponds to 3 different types of users. Each user can be either patient, doctor or lab. Users have the ability to modify the details of their profile.

2.2.6. Authentication:

Validates the login and the registry of users into the database. Secures access to the database by the users by means of tokens, etc.

2.2.7. Access list:

Module which manages the access list of a particular patient. When a patient gives access to a doctor then the access is visible to both doctor and patient.

2.2.8. User:

Central module that interacts with all other modules

2.3. Technology/ Tools Used:

2.3.1. Client-side / Front-End:

Display of the application and the user interface / interactions

- Web App's front-end will be in HTML5, CSS, JavaScript

2.3.2. Server-side / Back-End:

- Web App's backend in PHP

2.3.3. Database:

- Azure Table Storage
- Azure Blob storage



3. Detailed design:

3.1. Module APIs:

3.1.1. Registration and Authentication:

3.1.1.1. Registration

- **user_registration_details(form):** Registration request with form data having fields such as username, password, role, etc.
- **validate_details(form):** Checks for any inconsistencies in the form data received in request.
- **register(form):** To finally save the validated data in to the database.

3.1.1.2. Login

- **login(form):** Login request with username and password as parameters, filled in web page form by user.
- **validate_details(form):** Validates the form data sent by user for things such as, empty form fields, invalid characters sent.
- **authenticate(form):** Checks if the username, password is in database or not, authenticate accordingly.

3.1.2. General User Information:

3.1.2.1. Update Profile

- **update_details(form, user_session):** Receives request to update profile information for user with form data as parameter and user_session variable to authenticate the user.
- **validate_details(form):** Validates the form data sent by user for things such as, empty form fields, invalid characters sent.
- **save_details(form):** Save the details to database if the request is a valid one.

3.1.2.2. Get Profile Data

- **fetch_details(form, user_session):** Receives request to get profile information for user. Authentication is done by user_session variable.
- **get_details():** Gets the profile data from database.

3.1.3. Patient:

3.1.3.1. Search Doctor/Labs

- **search_user_by_username(username):** Receives request with username as parameter.
- **get_user(username):** Interacts with DB to give back search results for given username

3.1.3.2. Give access to Doctor/Labs

- **give_access(username, user_session):** Receives request with username and user_session as parameter.
- **add_access_relationship(username, patient_username):** Adds access relation from username to patient_username in DB.

3.1.3.3. Search tags

- **search_tag(text):** Receives request with text as parameter.
- **get_tag(text):** Interacts with DB to give back search results for given text in Tags table.

3.1.3.4. Edit tags

- **edit_tag(old_tag, new_tag, user_session):** Receives request with old_tag, new_tag and user_session as parameter.
- **replace_tag(old_tag, new_tag):** Updates the old_tag text with new_tag text in DB.

3.1.3.5. Fetch Timeline

- **fetch_timeline(user_session):** Receives request with user_session as parameter.
- **get_records():** Gets all the records and related comments to generate timeline.

3.1.4. Message:

3.1.4.1. Send Message

- **initiate(username, message, user_session):** Receives request with username, message and user_session as parameter.
- **send_message(username, message, user_session):** Saves the message entry into DB.

3.1.4.2. Get Messages

Get all messages exchanged between 2 users(Patient, Doctor).

- **initiate(username, user_session):** Receives request with username and user_session as parameter.

- **get_messages(user1, user2):** Gets all the messages exchanged b/w user1 and user2 in chronological order.

3.1.4.3. Get Active Conversations

List the users with which the requesting user had conversations.

- **initiate(user_session):** Receives request with user_session as parameter.
- **get_active_conversations(user1):** Gets the users which have messages exchanged with user1 and display them in chronological order w.r.t to last message exchanged time.

3.1.5. Lab:

3.1.5.1. Upload Reports

List the users with which the requesting user had conversations.

- **upload_image(image, patient_id, user_session):** Receives request with image, patient_id and user_session as parameter.
- **create_container_and_upload():** Upload the image to Blob storage.
- **update_timeline():** Stores new report in Timeline table in DB.
- **add_notification():** Generates a notification for this new report in Notification table in DB.

3.1.6. Doctor:

3.1.6.1. Search Patient

Search patients by username

- **search_patient(patient_username, user_session):** Receives request with username and user_session as parameter.
- **validate_access(patient_username, doctor_username):** Validates access based on validations such as username exists, there is an access relationship b/w doctor and patient.

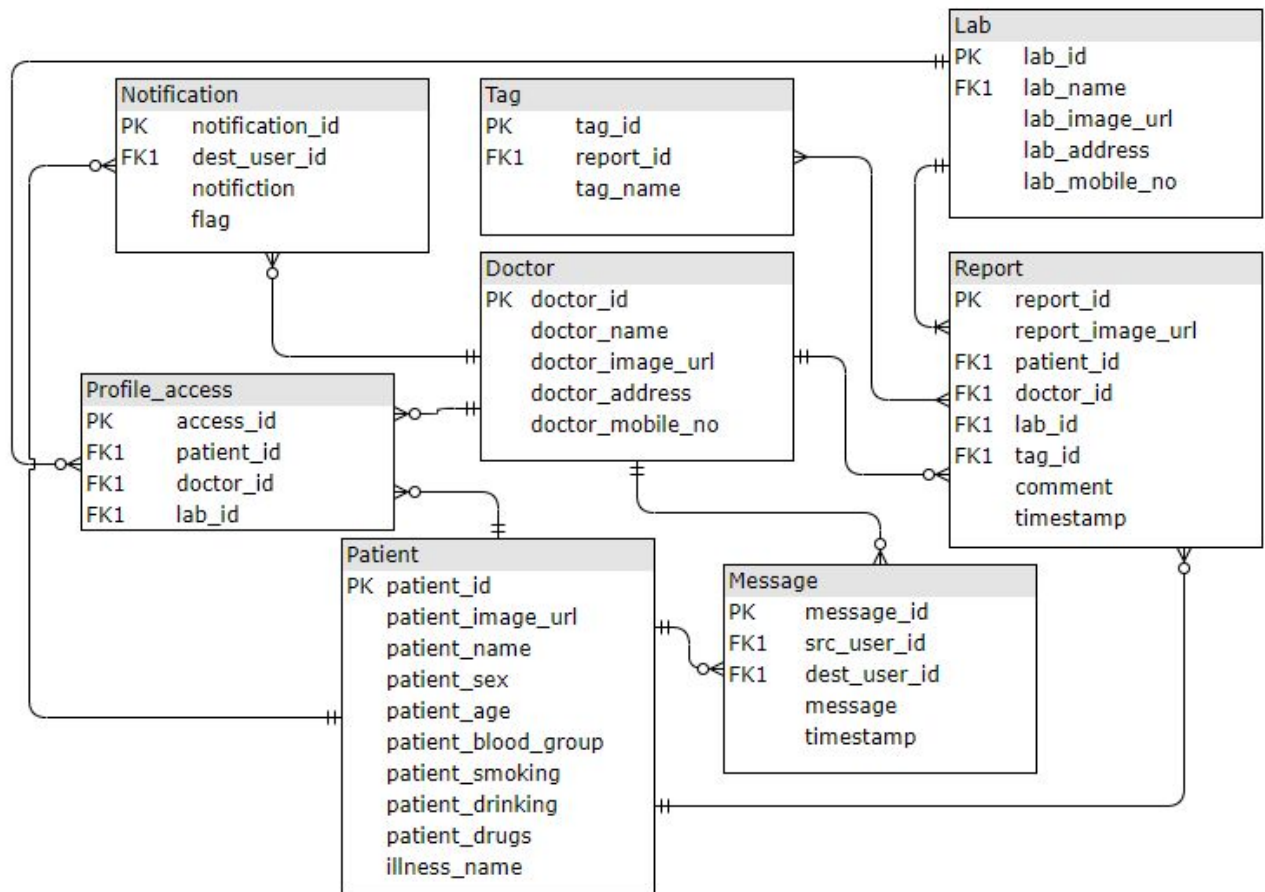
3.1.6.2. View Timeline of Patient

- **view_timeline(patient_username, user_session):** Receives request with patient username and user_session as parameter.
- **get_records(patient_username):** Finally checks for permissions and fetches record from DB to be shown.

3.1.6.3. Write comments on reports

- **write_comment(record_id, comment, user_session):** Receives request with record_id, comment text and user_session as parameter.
- **add_comment():** Adds comment to database.

3.2. Database Design:



3.2.1. Data characteristics:

As mentioned earlier, the data will be stored in an Azure SQL database with images in Azure blob database. The data will be kept according to the tables given in the above database schema.

3.2.2. Database Functionality - CRUD:

- **Create(collection, document)** : Insert document in the collection.
- **Retrieve(collection, fields)** : Retrieve information from the collection with the fields specified.
- **Update(collection, filter, field, new_data)** : Update data corresponding to the given field in the documents specified by the filter in the collection with the provided new_data.
- **Delete(collection, filter, field)** : Delete fields from documents specified by filter in the collection

3.2.3. Sub-databases:

3.2.3.1. Patient Details

Stores personal profile details related to patient.

3.2.3.2. Doctor Details

Stores personal profile details related to doctor.

3.2.3.3. Lab Details

Stores personal profile details related to patient.

3.2.3.4. Report Details

Stores details related to the report. It contains mapping to various tables such as patient, lab, doctor, tag etc.

3.3. Screen Layouts:

Register

Register page

Name

Username

Password

Register as ▼

Register

patient

doctor

lab

Login

login page

username

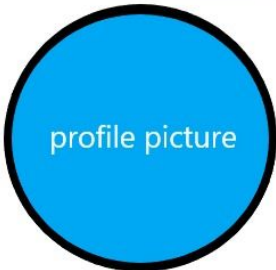
password

Select Role

login

Patient Profile

profile	timeline	access list	messages	notifications
---------	----------	-------------	----------	---------------



patient name

logout

Age: _____

Sex: _____

Blood Type: _____

*Allergies: _____

*Habits (Smoking/ drinking/ drugs): _____

*Family History of illness: _____

*Past major diseases: _____

*Past surgeries: _____

* = optional

Patient accessibility list

profile

timeline

access list

messages

notifications

add a new doctor :

Doctor name

Doctor id

Add doctor

list of doctors having access:

doctor name1	doctor id 1	delete
doctor name2	doctor id2	delete
doctor name3	doctor id3	delete
doctor name4	doctor id4	delete
⋮		

Patient timeline

profile

timeline

access list

messages

notifications

Report Image

Lab name

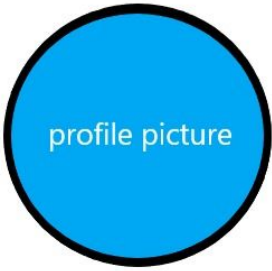
tags

prescription/
comments
by doctor:

search within tag:

search

Lab Profile

Profile	Upload report
<div></div> <div>Lab name</div> <div>logout</div>	<div>Address: _____</div> <div>Phone/mobile: _____</div> <div>* = optional</div>

Lab Upload:

Profile	Upload report
<div><div>select image</div><div>patient id</div><div>doctor id</div><div>tags</div><div>Upload</div></div>	

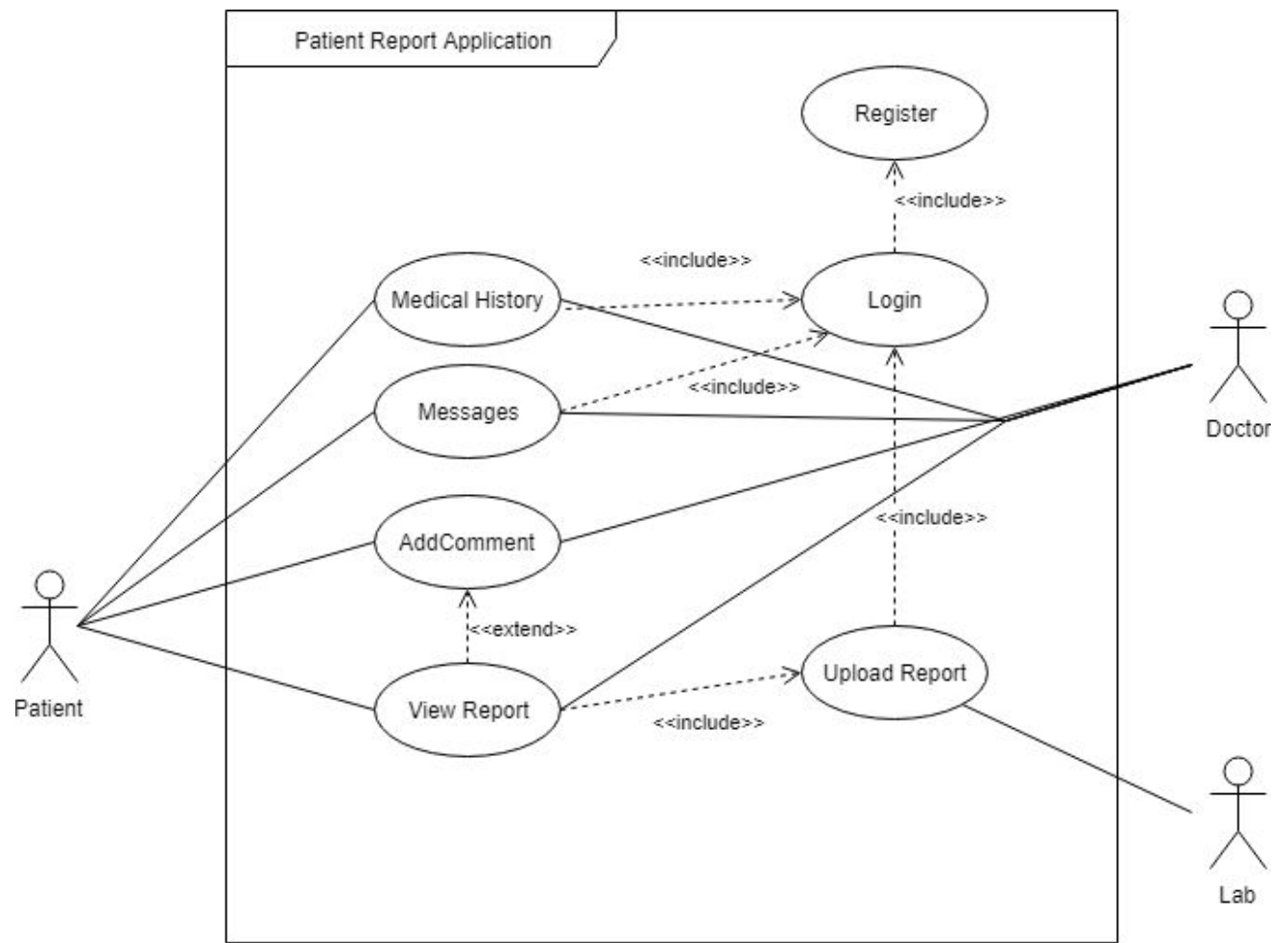
Doctor Profile

Profile	notifications	Accessible patients	messages
<div>profile picture</div> <div>Doctor name</div> <div>logout</div>		<div>Address: _____</div> <div>Phone/mobile: _____</div> <div>* = optional</div>	

Doctor accessibility list

Profile	notifications	Accessible patients	messages															
<table border="1"><tr><td>Patient_name1</td><td>patient_id1</td><td>view_profile</td></tr><tr><td>Patient_name2</td><td>patient_id2</td><td>view_profile</td></tr><tr><td>Patient_name3</td><td>patient_id3</td><td>view_profile</td></tr><tr><td>Patient_name4</td><td>patient_id4</td><td>view_profile</td></tr><tr><td colspan="3">• • •</td></tr></table>				Patient_name1	patient_id1	view_profile	Patient_name2	patient_id2	view_profile	Patient_name3	patient_id3	view_profile	Patient_name4	patient_id4	view_profile	• • •		
Patient_name1	patient_id1	view_profile																
Patient_name2	patient_id2	view_profile																
Patient_name3	patient_id3	view_profile																
Patient_name4	patient_id4	view_profile																
• • •																		

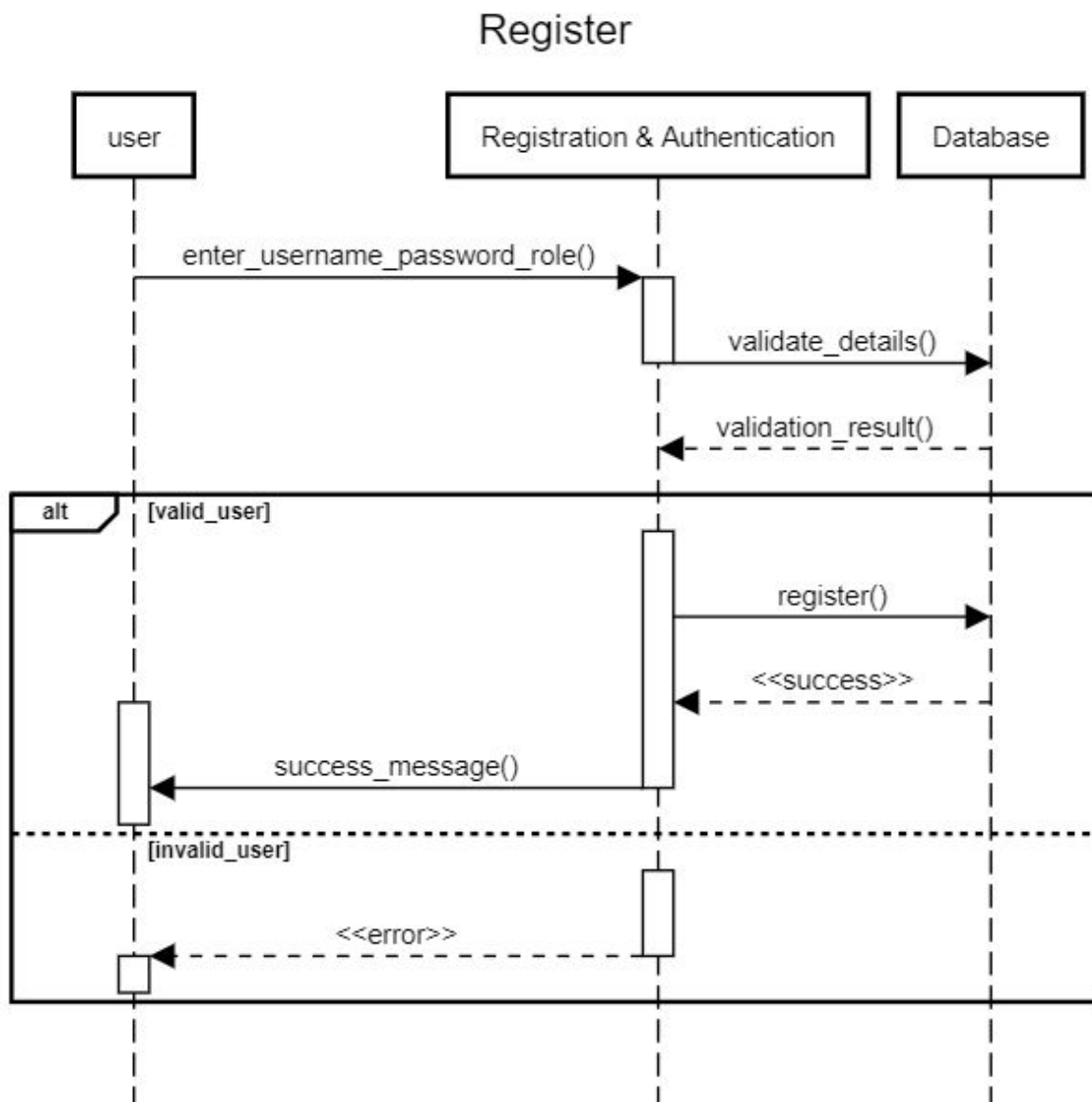
3.4. Use Cases:



Use Case Diagram

3.4.1. Sequence Diagrams and High Level Code:

3.4.1.1. User Registration



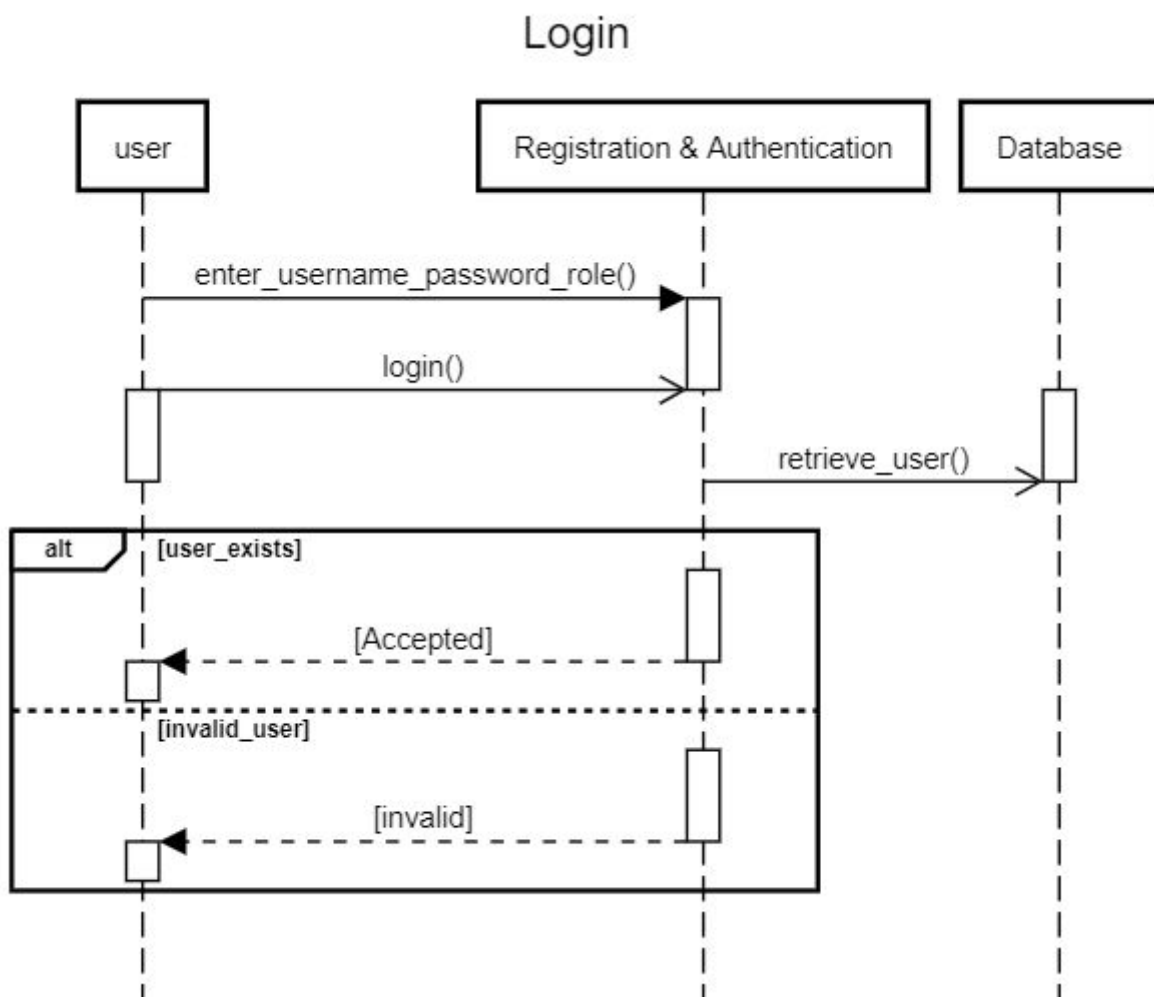
```
registration()
{
    request = request_registration()
    isValid = authenticate_registration()
    if (isValid == True)
    {
        add_user(request)
        data = registration_response(request)
    }
}
```

```

    display_data(data)
}
else
{
    show_error_screen(Error)
}
}

```

3.4.1.2. User Login



```

login()
{
    request = request_login()
    isValid = authenticate_login()
}

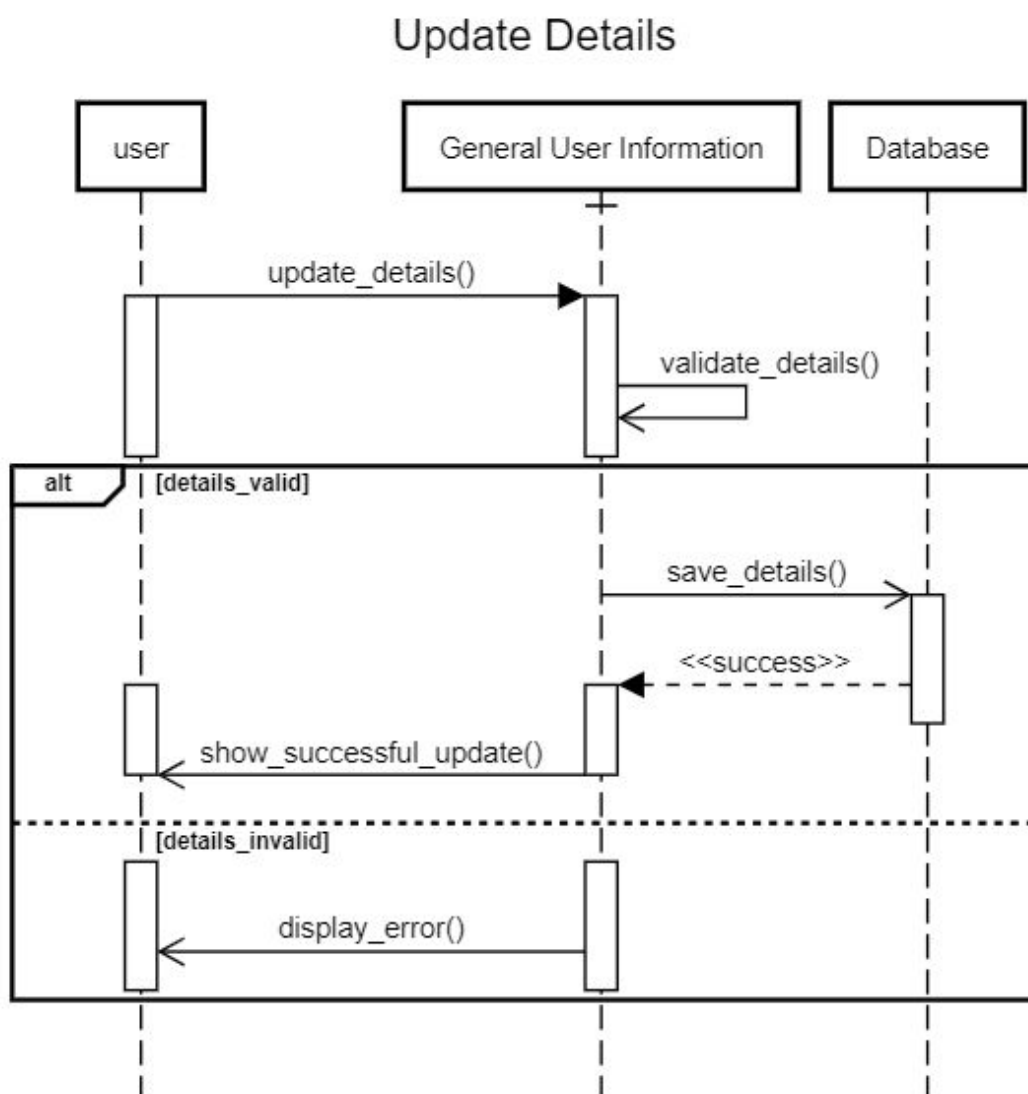
```

```

if (isValid == True)
{
    update_databse(request)
    data = login_response(request)
    display_data(data)
}
else
{
    show_error_screen(Error)
}
}

```

3.4.1.2. Update profile details:

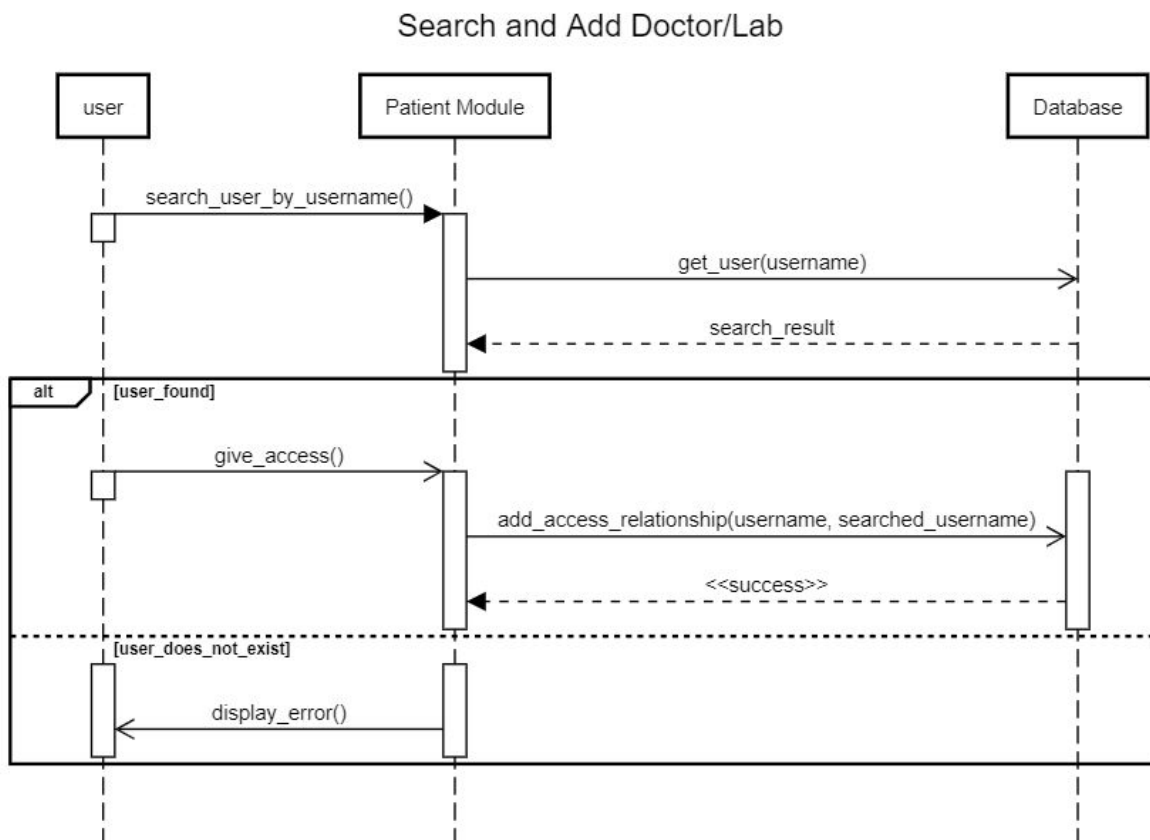


```

update_profile(username)
{
    data = request_user_data()
    if validate_data(data)
    {
        for field in profile:
            profile[field] = user[field]
            update_profile()
        }
    else
    {
        show_error_screen(Error)
    }
}

```

3.4.1.3. Search and Add Doctor/Lab



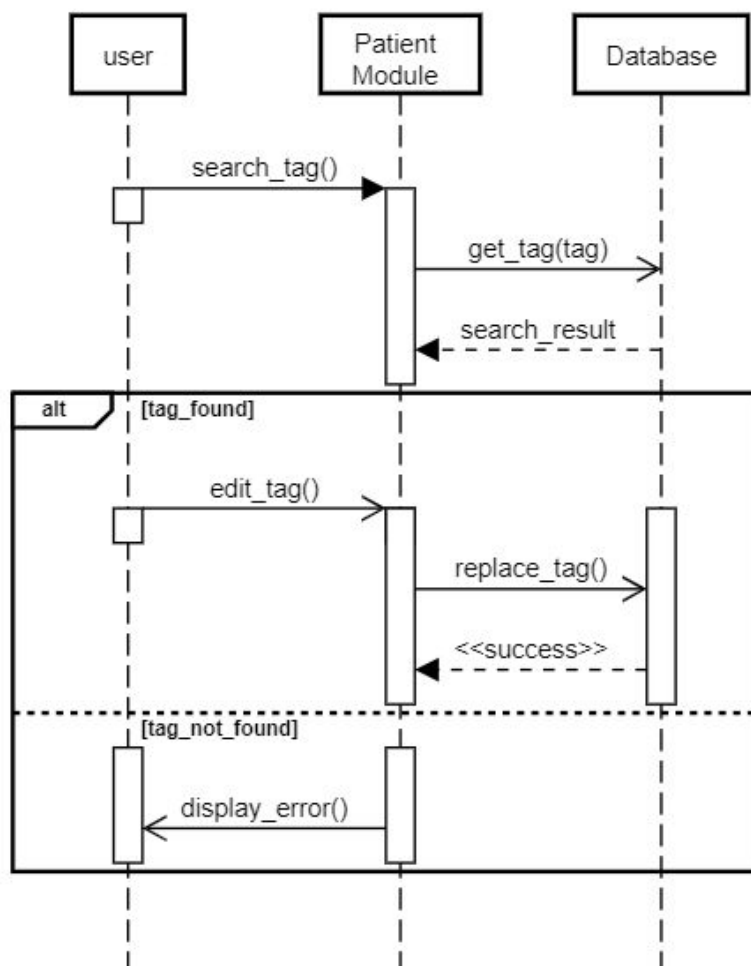
```

add_user(username1)
{
    username2 = request_username_to_add()
    if (search_username(username2)==found)
    {
        add_access_relationship(username1, username2)
    }
    else
    {
        show_error_screen(Error)
    }
}

```

3.4.1.4. Search and Edit tags

Search and Edit tags

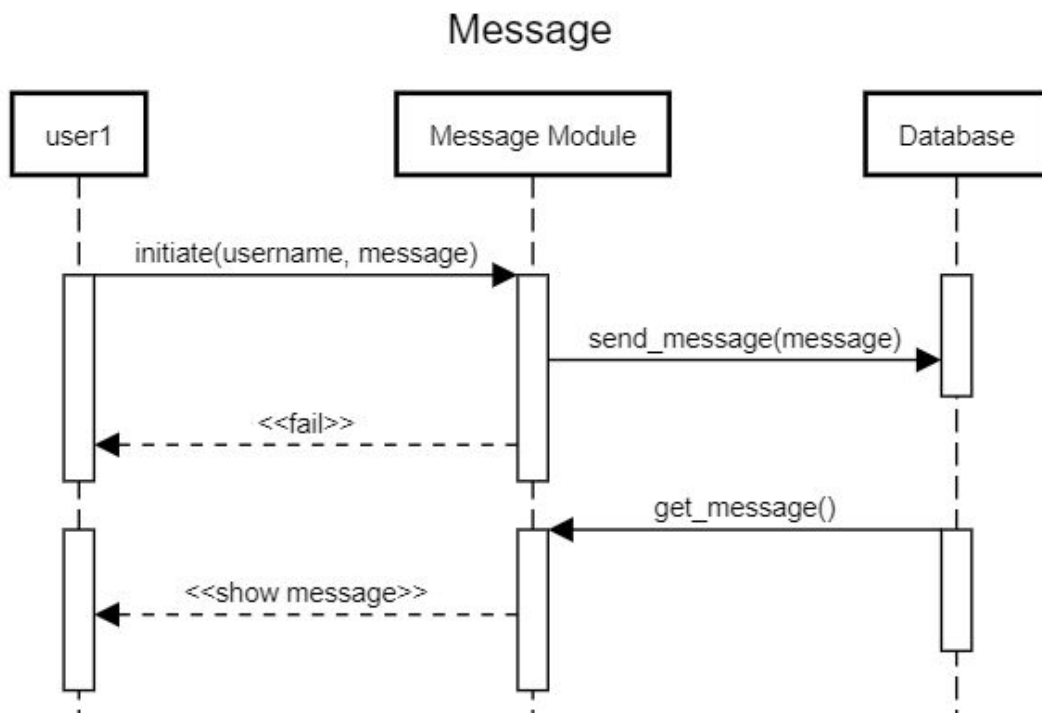


```

edit_tags(username)
{
    tag = request_tag_to_search()
    if (search_tag(tag)==found)
    {
        if (new_tag is not NULL)
        {
            replace_tag(tag, new_tag, username)
        }
    }
    else
    {
        show_error_screen(Error)
    }
}

```

3.4.1.5. Messaging service



```

messages(username1, username2)
{

```

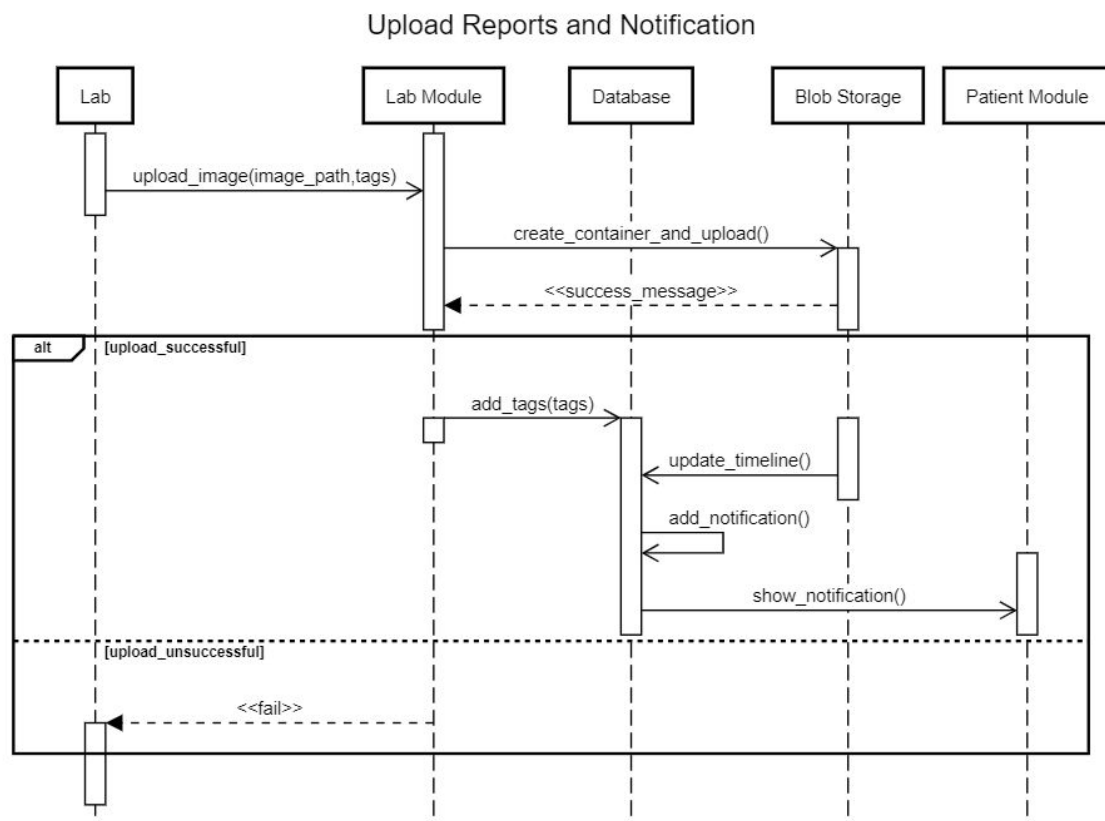


```

data = recieve_message(username2)
display_data(data, username1)
if (initiate_send()==true)
{
    send_message(username2)
}
}

```

3.4.1.5. Upload report and Send Notification



```

upload_report(username1, username2)
{
    path = get_image_path()
    if (path is not NULL)
    {
        blob = initiate_blob(username2)
        container = create_container(path)
        blob.add(container)
    }
}

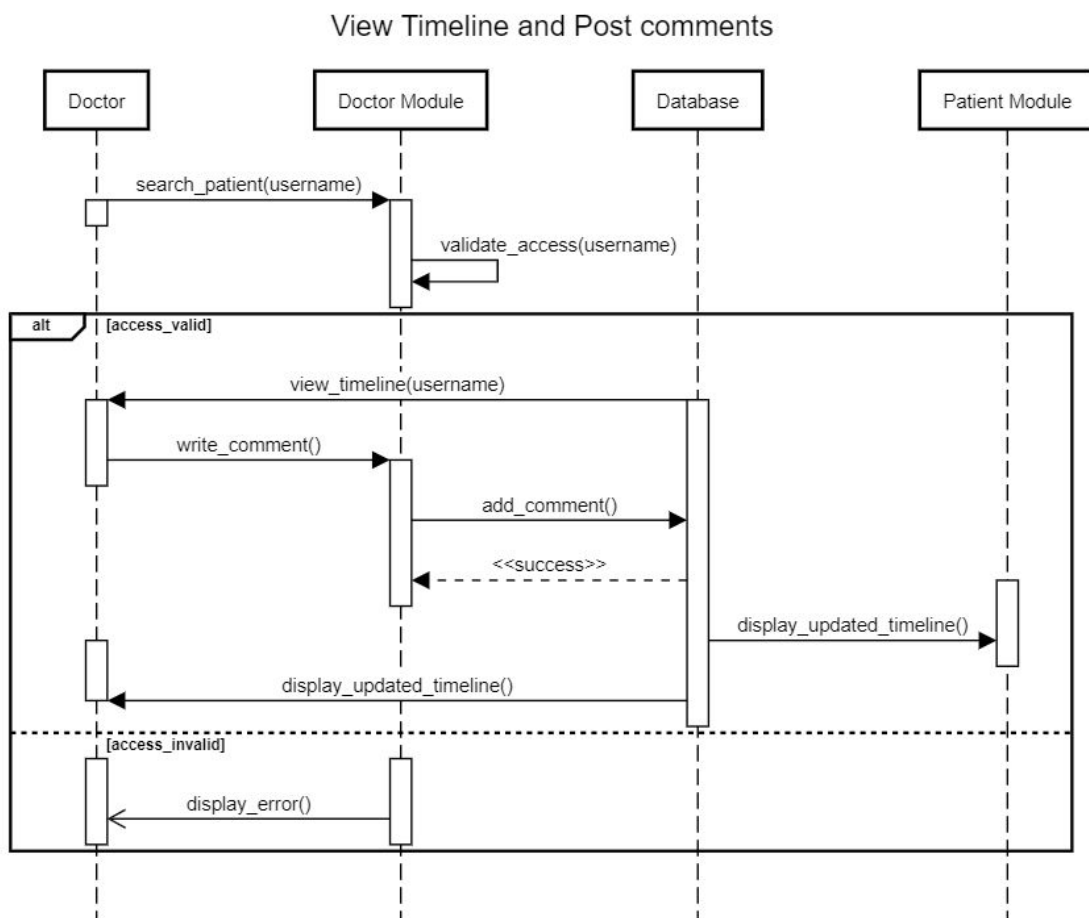
```

```

        upload_blob(username2, blob)
        display_data(username1)
        send_notification(notif, username2)
    }
    else
    {
        show_error_screen(Error)
    }
}
send_notification(notif, username)
{
    set_notification(notif, username)
    display_notification(username, notif);
}

```

3.4.1.6. View timeline and post comments



```
view_timeline(username1)
{
    search_patient(username2)
    if has_access(username1,username2)
    {
        view_timeline()
        request = comment_request()
        add_comment(request, username2)
        display_data()
    }
    else
    {
        show_error_screen(Error)
    }
}
```

4. Deployment Design:

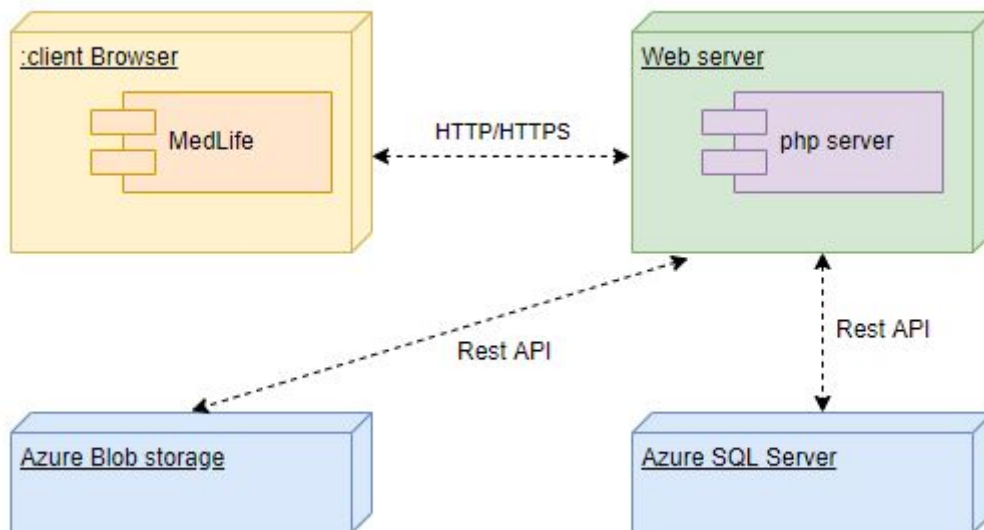


Fig: Deployment Diagram

4.1 Environment:

- The application development environment will be Sublime Text.
- Any operating system with popular browsers such as Chrome or Firefox with Javascript enabled will be used as target
- The website will require access to internet on the system.
- The project will use php as backend server, Azure SQL server for dB and Azure blob service for image dB.

4.2 Version Control:

- Version Control System - Git
- Platform - GitHub or BitBucket
- Git connected to Azure. So any commit to git automatically updates deployment at Azure.

4.3. Testing and Debugging:

4.3.1. Functionality Testing

- **Actions:** Check that all the buttons and input fields work like intended, in various scenarios.
- **Inputs:** Check all the input/form data is successfully submitted and processed at the webserver.

- **Database:** Check if all queries and updates are consistent with the integrity constraints.

4.3.2. Usability Testing

- **Navigation:** Ensure that navigating through the website is intuitive and familiar.
- **Content:** Verify that the content is what you intended to be. Check spellings, fonts, colors etc. There should be no surprises.

4.3.3. Interface Testing

Ensure that any disconnection between the webserver and the website is handled, reported and logged properly.

4.3.4. Compatibility Testing

- **Device compatibility:** Test the website to be compatible with all the devices with targeted browsers.
- **Server compatibility:** Test the server application to be compatible with Windows as well as Linux servers with appropriate changes.

4.3.5. Performance Testing

- **Website Stress Testing:** Verify that website does not break or behave rampantly on invalid inputs or loss of internet.
- **Load Testing:** Verify that the server reacts normally with high number of users putting load on the database and specific pages by posting or fetching huge data.

4.3.6. Security Testing

- **Authentication:** Ensure that profiles can only be accessed after logging in.
- **Logging:** All transactions and error messages must be logged so that security breaches can be studied later.