SOFTWARE DESIGN DOCUMENT

for

Vaccine management System

September 2021

1. INTRODUCTION

This software design document (SDD) is a formal architectural blueprint for Vaccine Management System. The layout is structured to help the reader recognize the requirements imposed upon the system and how these requirements are being met.

1.1. Purpose

This document describes all the standards and design of VMS (Vaccine Management System) to be used by the developers to follow in the implementation phase. The SDD is an important reference not only for developers, but also for the testing programmers and editors of documentation.

1.2. Scope

VMS is software that needs to be deployed on a central server used by the admins. VMS will be used to run vaccine campaigns and will allow government agency to manage their campaigns. The users of the system are the people who need to get their vaccines. They should be able to book their appointments using the web interface of the software.

2. ARCHITECTURAL DESIGN

The applications web Interface will be built using C# and .NET whereas the Database will use Mongo DB.

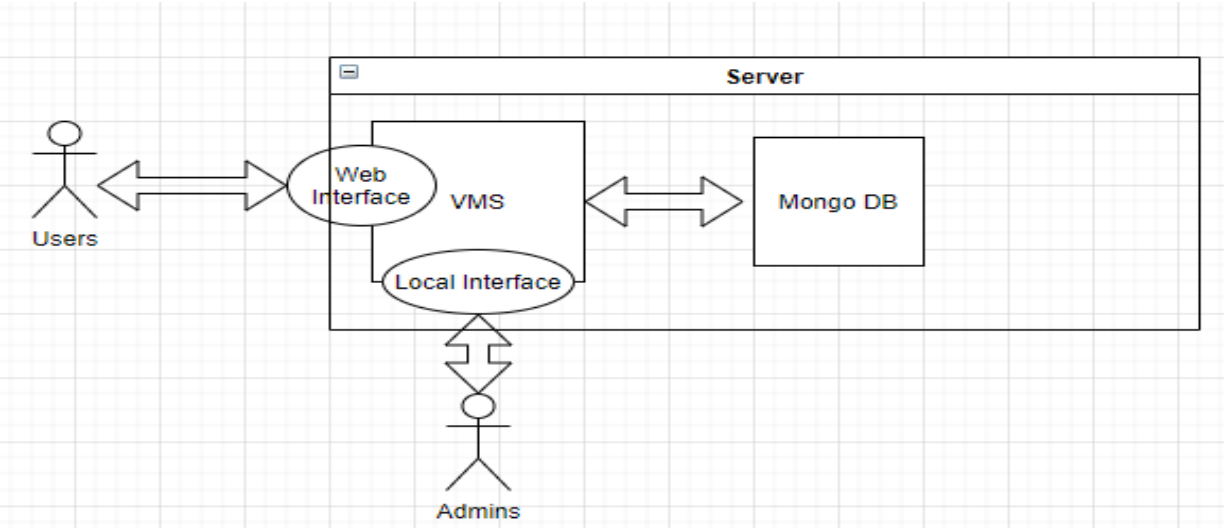


Figure 1: High level Architecture

The system also has two Crypto Keys that will be generated when the system starts for the first time and are stored in the specific files that are password protected.

1: Encryption Key: This key will be used to encrypt all the data that goes and comes out of the database.
2: Certificate Key: This key will be used to sign all the certificates.

Using two key ensures that if one system is compromised the other is still secure.

The users of the system namely admins and the users access it through web interface and local interface using two different URLs respectively. The admin URL will be only available on the Government agencies network and direct web access needs to be barred and is incumbent on the agency to handle this.

3. SYSTEM DESIGN

2.1 VMS Modules:
The VMS comprises of the following components:
1: **Authentication**
The purpose of this module is to manage Signup and login for both users and admins. It will also implement Multi Factor Authentication using .NET core Identity library. The current MFA only consists of 2FA whereby the admin will receive a code on their registered email id.

This component also manages authorization and will only allow users who have verified their emails. For email verification user manager library of the asp.net core will be used. The Authentication component will send a link on the email to verify the users and admins.

It also needs to implement password criteria. The password must check all the criteria mentioned below:
1: Must be greater than 8 characters.
2: Must have numbers.
3: Must have capital letters.
4: Must have one special character.

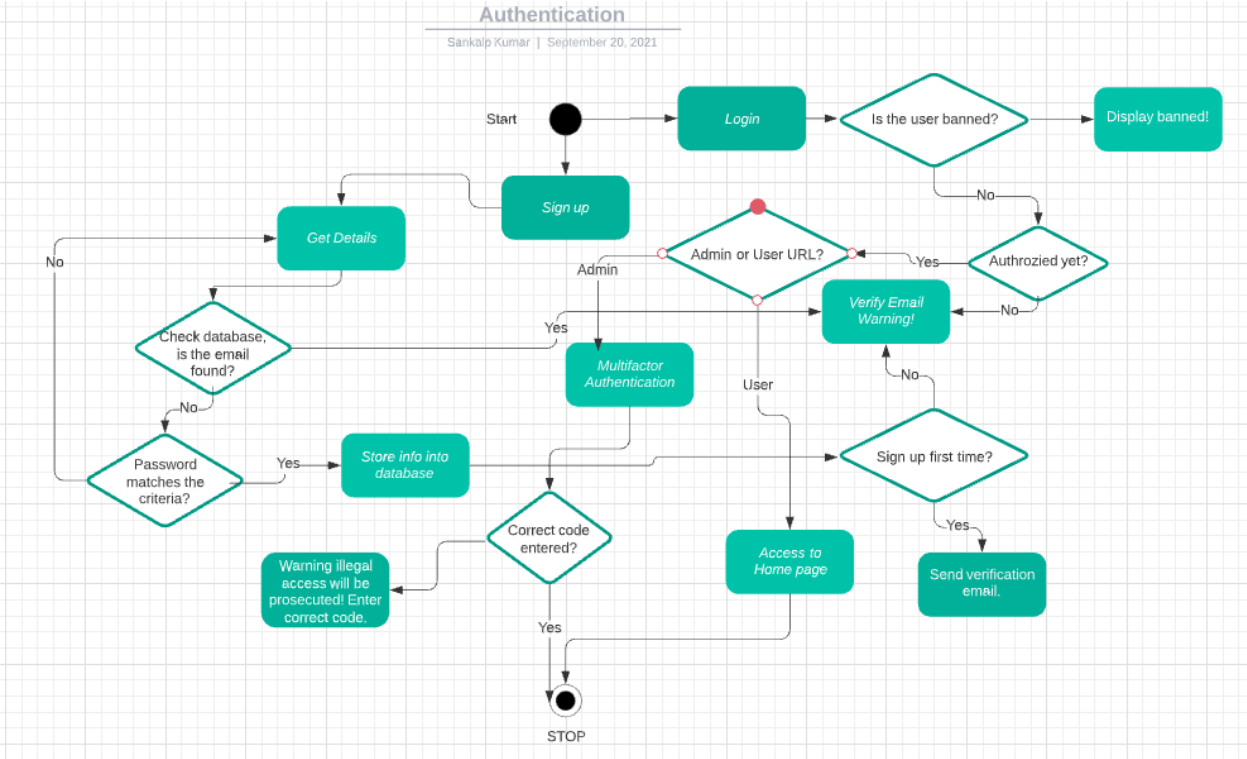


Figure 2: Flow Chart for Authentication

2**: Key management layer**
This layer manages all kinds of keys be it password or crypto keys. This is responsible for mandating password policies and recreating keys.

The module regenerates keys and re-encrypts the entire database on a yearly basis. Since, this could lead to all user's certificate getting invalidated suddenly in one shot, there could be a scenario where all users would want to renew their certificates, and this could lead to high load on server. This being a performance problem could be solved in a later release whereby the re-encryption happens in phases.

It also mandates admins to change their key on quarterly basis by running checks on date time stamps stored in databases.

3: **Database Layer**
The layer interfaces with the database and runs all transaction needed between the application and database.
Following are the function that the module must implement:
1: setUserDetails: sets the user details upon signup.
2: getUserInfo: takes input one or more fields to get from the database.

The setUserDetails also must implement the following functionality:
1: Hash the password using SHA256.
2: Add salt to the password.
3: Use Encryption key to encrypt all data before storing it in the database.

The encryption will be done using AES256-CBC using openssl library.

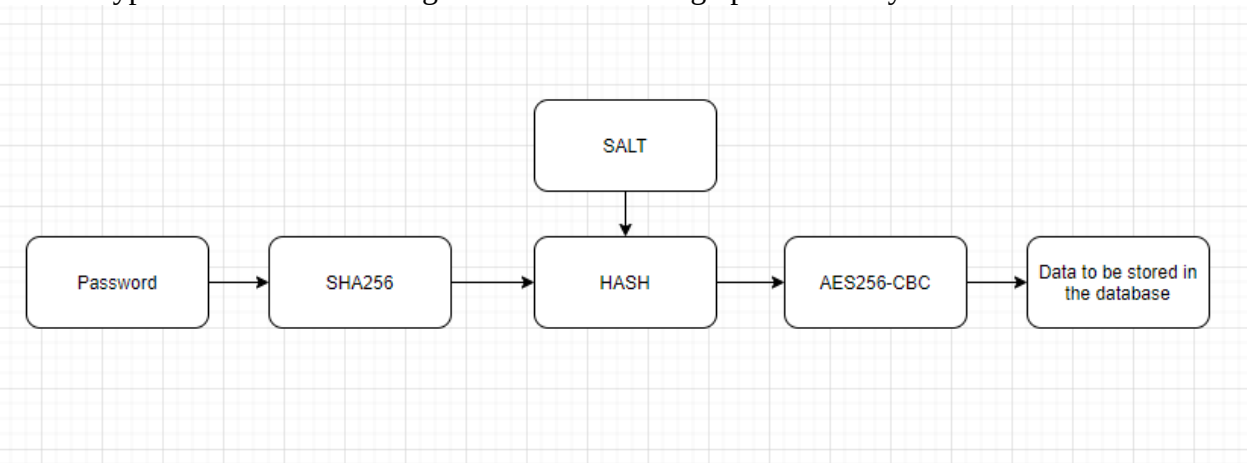


Figure 3: Storing the password

4: **Usage limit layer**
The layer sees over banning of Users and admins. We track each user of the system using their email and overusing the limit leads to banning of the user for a week. We do not use IP as this could be misused easily. Hence, the single source of truth for tacking users is their email address.

Following are the limits set for User:
1: If the user attempts to look for more than 5 zip codes for vaccine centers.
2: If the user tries to book more than one appointment for the same vaccine. Once this happens thrice it will lead to ban.

This layer runs a thread as well to check if the ban needs to be list. This thread is run every day at 7:00AM.

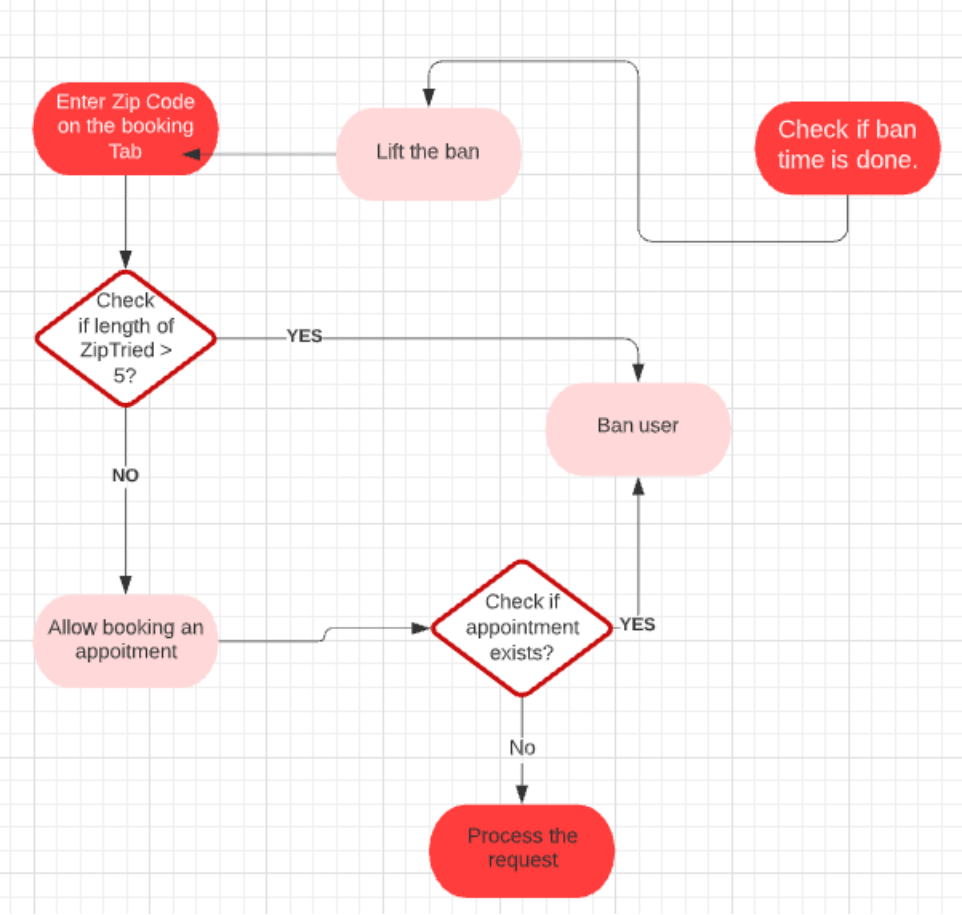


Figure 4: flowchart for user usage limit

Following are the limits to be set on Admins:
1: Admin attempts to create more than 3 vaccine campaigns in a day.

5**: Appointment attendance verification**
This layer generates an email just before the appointment and the user needs to enter this code in a screen that is only available via the admin URL of the application.

6: **Certificate generation Layer**
The Layer generates the certificate once the User has attended the appointment.
The certificate contains a hash created out of all user data using SHA256 and then encrypted using the Certificate key.

The algorithm to be used for encryption is AES256-CBC which is an approved symmetric key algorithm specified in FIP 140-2 published in the year 2019. We use openssl to generate keys and perform encryption and decryption. The paraphrase will change with each release. Due to security concerns the paraphrase has not been mentioned in this document.

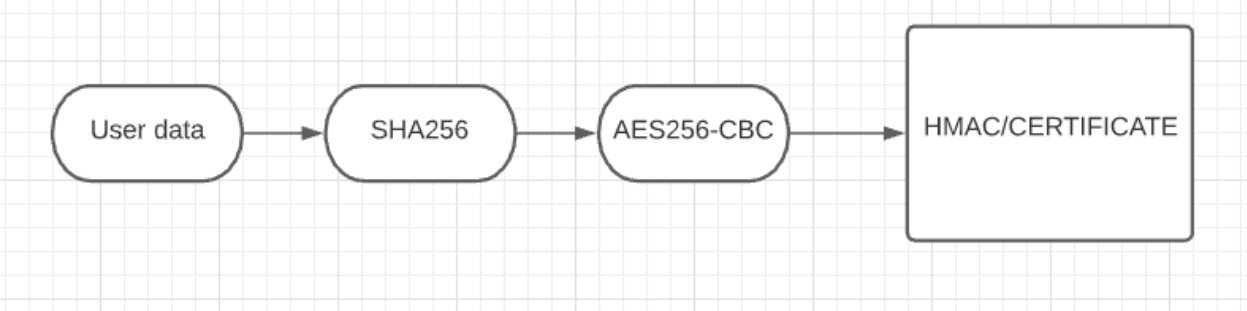


Figure 5: Certificate generation

7: **Logging Layer**
The logging layer maintains 10 files all of them compressed leaving out the current logging file. We build a wrapper called LOG that takes in multiple inputs to log to the files.
The log will note down all actions being taken by the user or admins.

The log rotates after writing to the 10th file. Each file will be no larger than 50MB. All these files must be password protected that could be set by the admins.

8: **Reminder:**
This will be a background thread called bgReminder. The thread is scheduled to run at 7:00am each day and send email reminders to all users who have appointment that day.

9: **Server time Check Layer**
This will be a background thread running and checking every 30 mins if the time is correct or not on the server. It will use the NTP version 4 and sync time from a NIST time server.

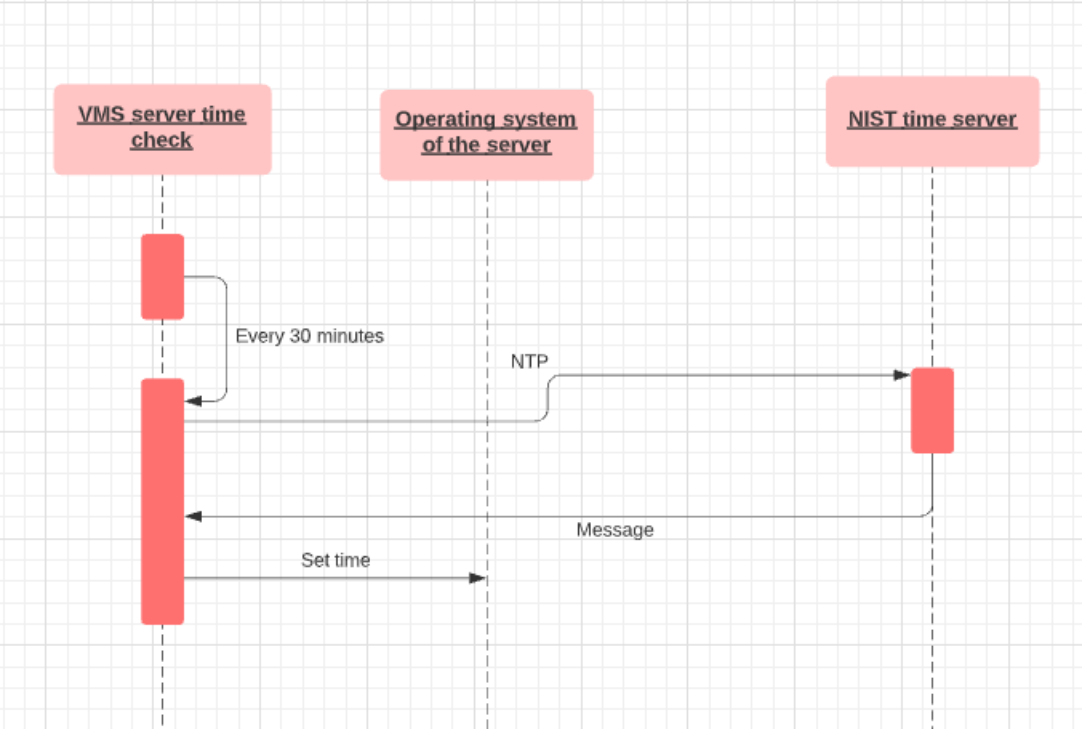
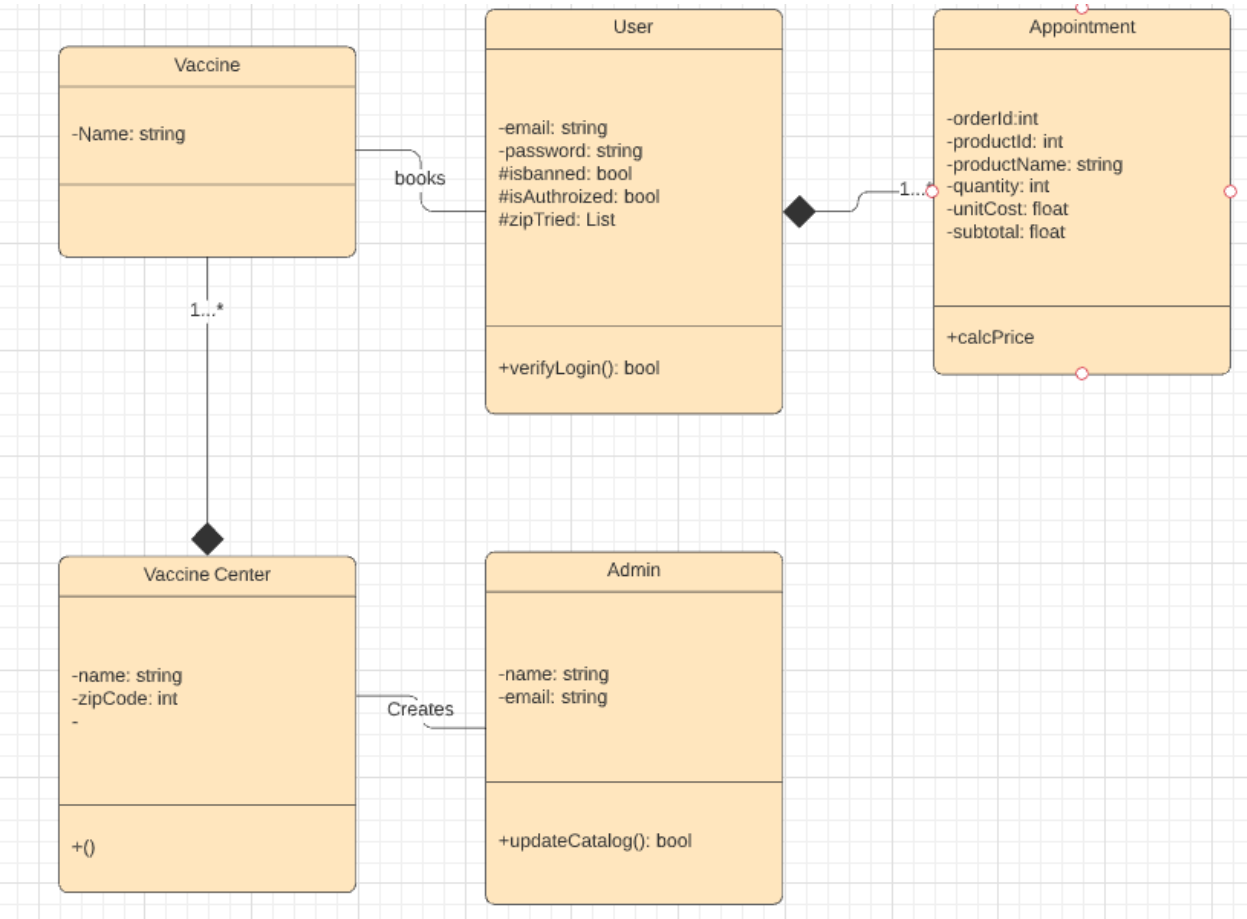


Figure 6: Sequence Diagram of time check layer

10: **Feedback**
Both admin and users interact with the feedback. Users add their feedback once they have attended the appointment. Admins can view the feedback. Searching or filtering through feedback will be done in future releases whereas for this release it will show the 10 latest feedback.

4. LOW LEVEL DESIGN
4.1 Classes and Methods:



<< Class diagram : Not yet complete. This will be updated soon with text. >>

4.2 Database Design
The plan is to use a non-relational database to store all the needs of our application.
We will use MongoDB as a database which is one of the most famous NoSQL databases.
MongoDB stores each table as BSON document and size of each of these documents is restricted to 16MB.

Following is the schema for the database:
User document:
{ _id : <objectId1>,
  name : “xyz”,

email : “xyz@gmail.com”
}

auth document:
{ user_id : <objectId1>,
isBanned: false,
banTime: new Date(),
isAuthroized: true
}

Sec document
{ user_id: <objectId1>
Pass : “qwerty”
Salt : 4526718
}

Appointment document
{ user_id:<objectId1>
name : “UMD Health Center”
zipCode : “27040”
}

feedback document
{ user_id:<objectId1>
feed : “Good campaign!”
}

admin document
{_id : <objectId2>
email : “admin@xyz.com”
}

asec document
{ admin_id: <objectId2>
Pass : “qwerty”
Salt : 4526718
}

aauth document:
{ user_id : <objectId2>,
isBanned: false,
banTime: new Date(),
isAuthroized: true
}

vaccineCenter document

{ _id : <objectId3>,
  name: “UMD health center”
  zipCode: “20740”

}

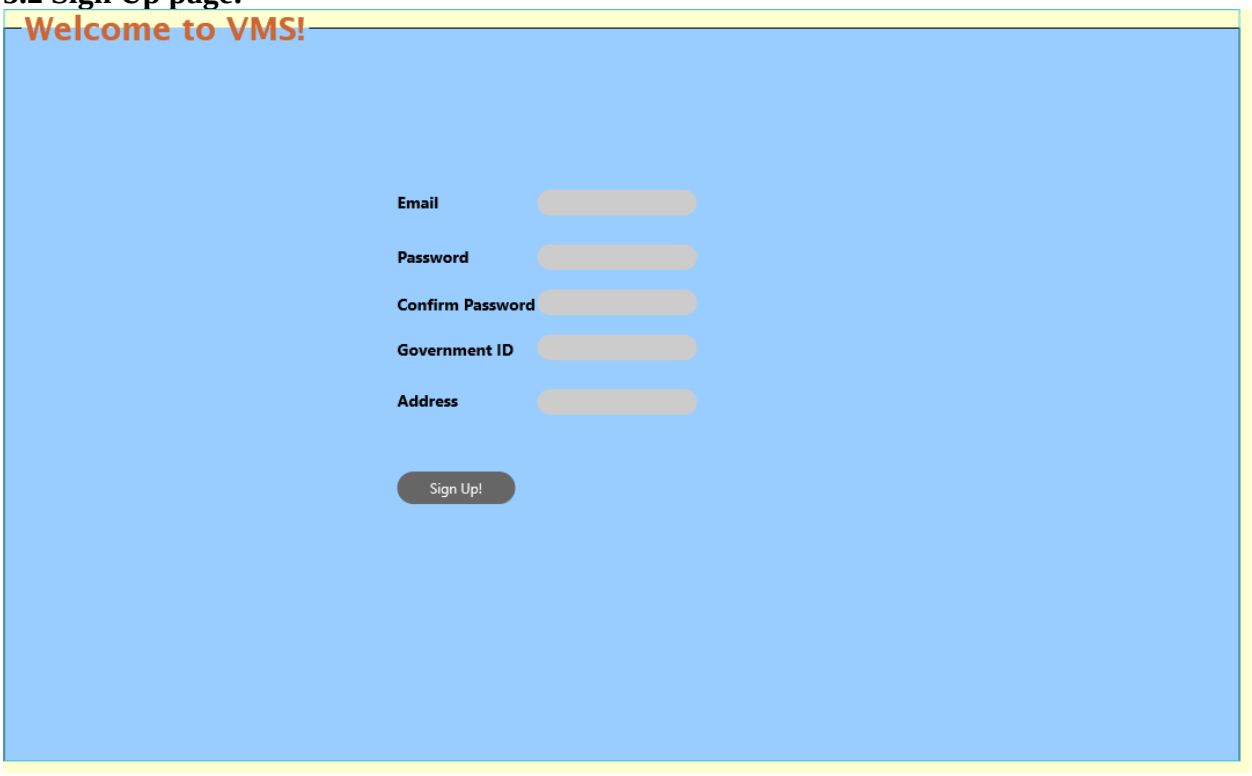
vaccine document
{_id : <objectId4>
 vaccineCenter_id : <objectId3>
 name: “AstraZeneca”
}

Certificate document
{user_id: <objectId1>
 vaccine_id : <objectId4>
 certHash: “8971923164127”
}

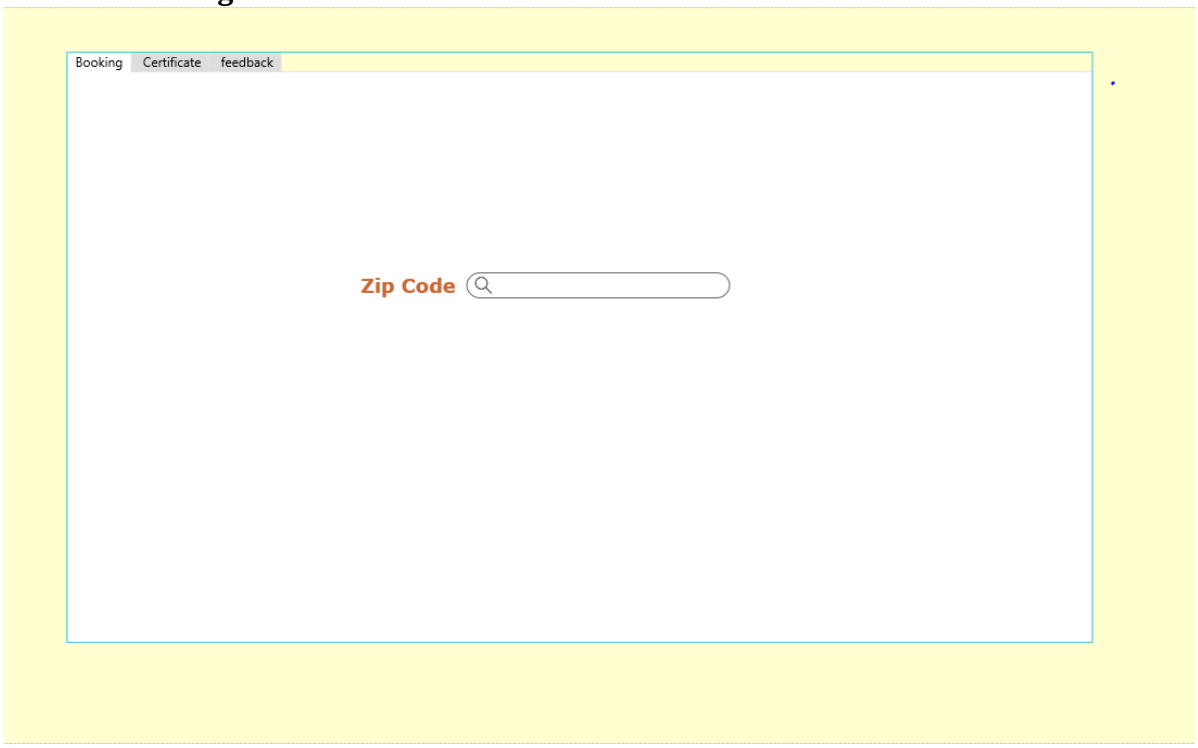
5. GUI

5.1 Login page:

5.2 Sign Up page:



5.3 User booking tab:

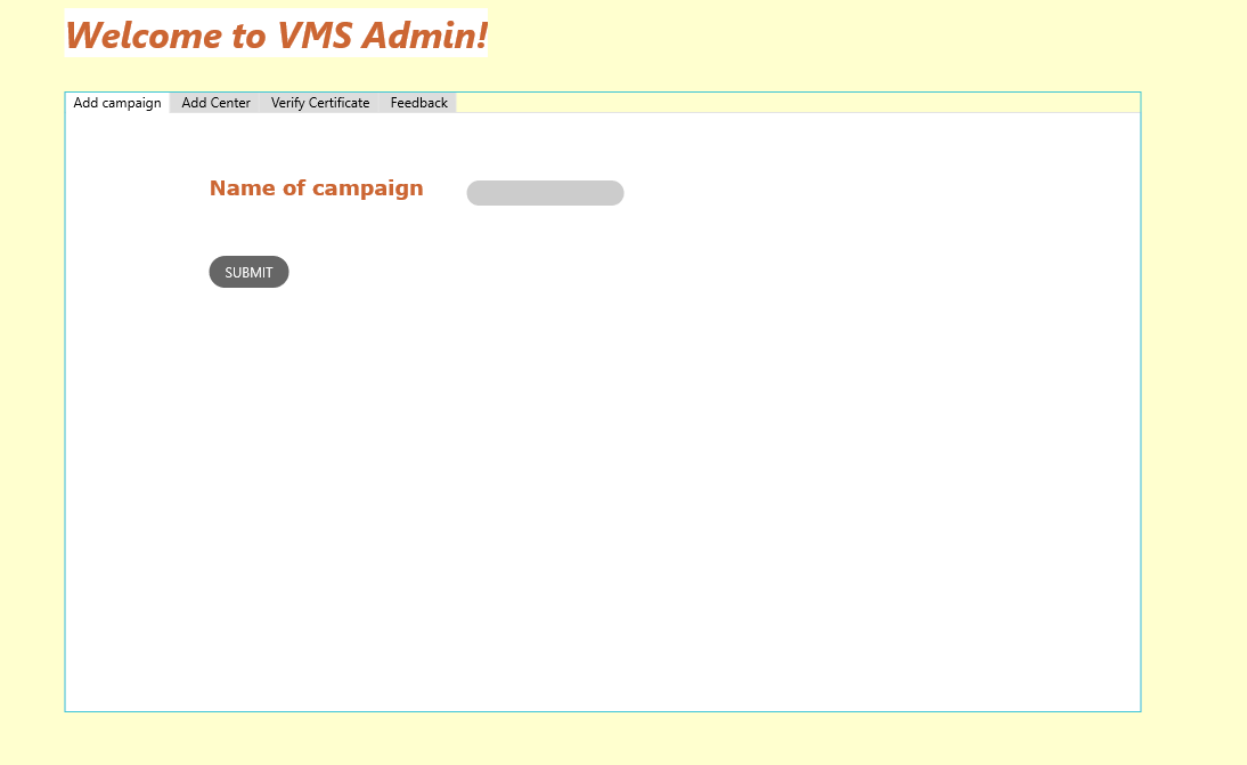


5.4 User feedback tab:

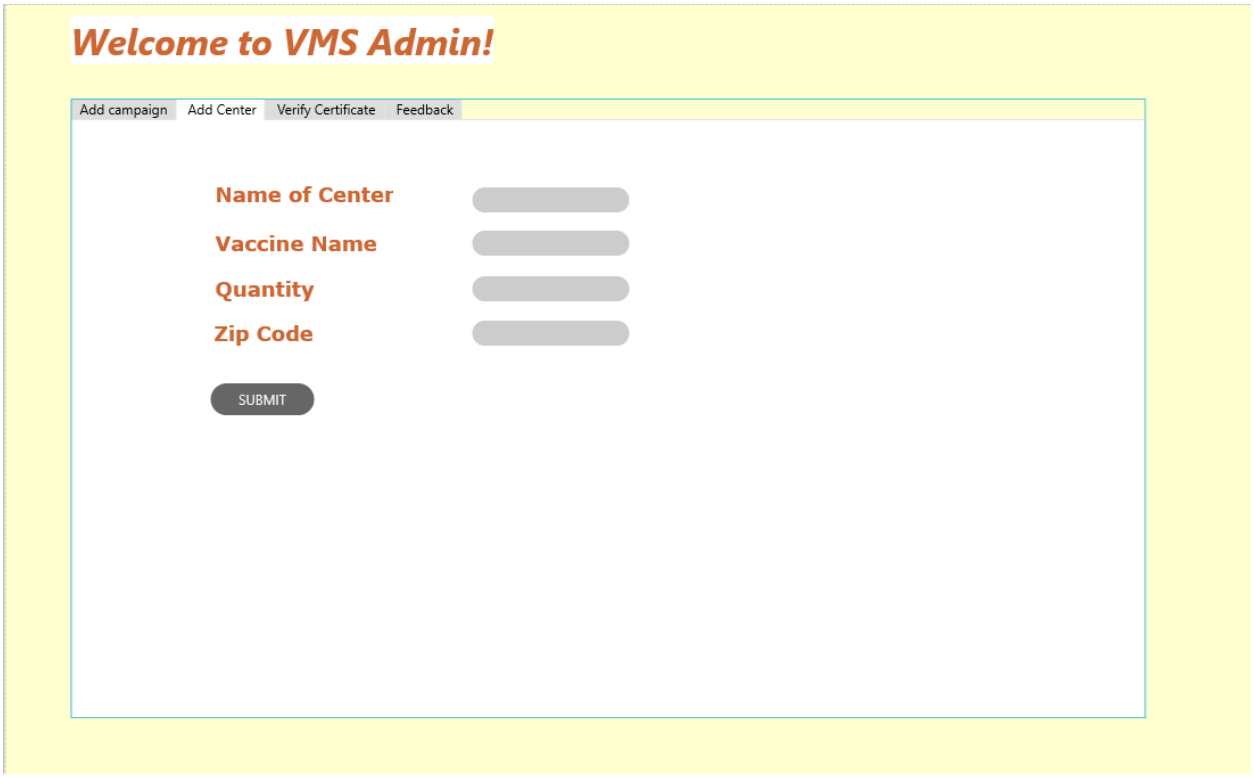
5.5 User certificate download tab:


Booking
Certificate
feedback
Covid
Download
General influenze
Download

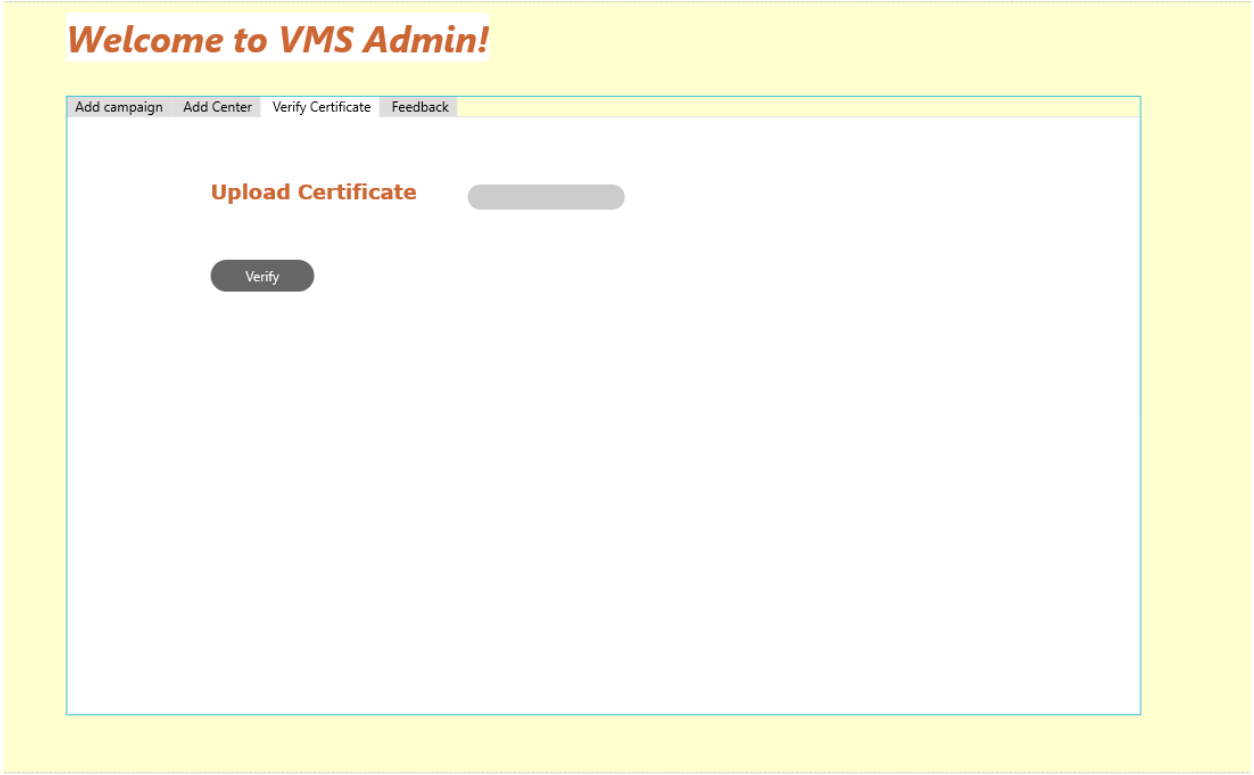
5.6 Admin Homepage:



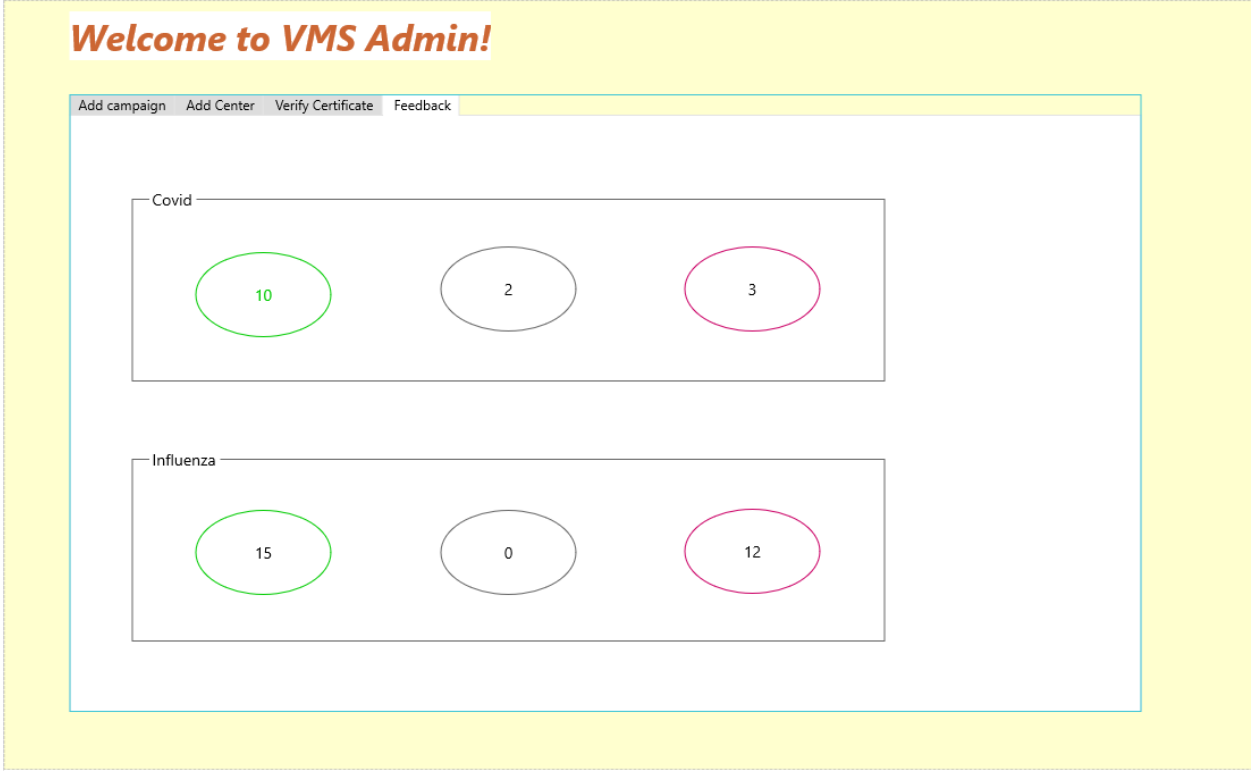
5.7 Admin Add center tab:



5.8 Admin Verification tab:



5.9 Admin View Feedback tab:



6. Testing
The project takes an approach called Test Driven Development in which all the test cases are covered before development. This, section covers the unit test cases for each of the modules described in the system design.

Unit test cases for authentication:
Signup
1: Leave one field blank and try signing up.
2: give an email id without the SMTP server name. Ex: Instead of xyz@gmail.com enter xyz.
3: Try signing up an already existing user with same information.
4: Try signing up an already existing user with the same email but different username or some other information.

Login
1: Try to login a user after getting banned.
2: Try to login a user that has not signed up.
3: Try to login a user that has signed up but not verified their email yet.
4: Try to login a user after ban has been lift.
5: Try to login using correct 2FA code.
6: Try to login using wrong 2FA code.
7: Try with a blank code.
8: Try admin logging into user.
9: Try user logging into admin.

Book an appointment
1: Try booking an appointment again.
2: Try booking an again at a different zip code.

Test reminder
1: Check if reminder is received when booking is made.

Banning and removing ban
1: Attempt getting banned

Change time
1: Change time and see if the system corrects it.