IFS4205 Information Security Capstone Project Group 3

Final Report

MediBook Medical Application

AY 2022/23 Semester 1

Oscar Lai Chun Ho	A0204697W
Alicia Tay	A0204813N
Daryl Hung Dejian	A0217127L
Isaac Lai Zile	A0216952B
Lim Jie Rui	A0217086A

1. Overview **4**

2. Roles **5**

3. Frontend **6**
3.1 Frontend Technologies 6
3.2 Frontend Deployment 6
3.2.1 Setup 6
3.2.2 Deploying React App to Apache Server 6
3.2.2 Running the Server 6
3.3 Frontend Code Structure 7
3.4 Frontend Security Implementation 8
3.4.1 AuthProvider 8
3.4.2 HTTPS 10

4. Backend **11**
4.1 Backend Technologies 11
4.2 Backend Deployment 11
4.3 Backend Security Implementation 11
4.3.1 Authentication 11
4.3.2 Permissions (Role-based access control) 11
4.3.3 Cross Site Request Forgery (CSRF) protection 12
4.4 Backend Testing 12
4.5 Backend Logging 13

5. APIs between Components **14**
5.1 Frontend and Backend 14
5.2 Backend and Database 18

6. Authentication **19**
6.1 Login subsystem 19
6.1.1 Implementation 19
6.1.2 Token 20
6.2 Two factor authentication subsystem 21
6.2.1 Implementation 22
6.2.2 Changing Authenticator: 24

7. Doctor Patient Flow Subsystem **26**
7.1 Flow Process 26
7.2 Security Implementation 29

8. Researcher k-anonymity Subsystem **31**
8.1 Process 31
8.2 Querying the data 32

9. IoT Crowd Sensing Subsystem **34**
9.1 IoT Technologies 34

9.2 Implementation 35
9.2.1 Interaction with Database 35
9.3 Security implementation 36

10. Database **37**
10.1 Implementation 37
10.2 Generating data 39
10.3 Security Implementation 40

11. DevOps **41**
11.1 Pre-commit 41
11.2 Continuous Integration (CI) 41

12. DevSecOps **42**
12.1 Software Composition Analysis (SCA) 42
12.2 Security application security testing (SAST) 43

13. Security Claims **44**
13.1 Server and Infrastructure Security Claims 44
13.2 Web Security Claims 44
13.3 Functional Security Claims 45
13.4 Internet of Things (IoT) Security Claims 46

14. Appendix **47**

1. Overview

This final report provides a detailed and extensive outline of each subsystem in our web service application. The report will be an extension to the system design and architecture diagram described in the Tool Assessment Report as well as the System Design Report. In each subsystem, the components, implementation and security concerns are explained in detail. Security features and claims of the web application are also discussed.

Introduction

Our web application is MediBook and can be accessed via https://ifs4205-group3-webapp.comp.nus.edu.sg/. MediBook is a medical facility application that simplifies the process for patients, doctors and researchers. What makes our medical application special is our use of IoT Crowd-Sensing Device and our patient-doctor flow that protects patient data privacy

Our application is divided into 4 different components; *frontend* which is responsible for user interface for user interaction with the server side, it is hosted on https://ifs4205-group3-webapp.comp.nus.edu.sg, *backend* which is responsible for storing and organising data and linking the frontend with the database, it is hosted on https://ifs4205-group3-backend.comp.nus.edu.sg, *database* which stores all essential data necessary for the web application to run, it is hosted on ifs4205-group3-database-i-.comp.nus.edu.sg. Lastly, we have our IoT component which uses a raspberry pi to count the crowd inside the medical facility. It ports the data straight to the backend via HTTPS.

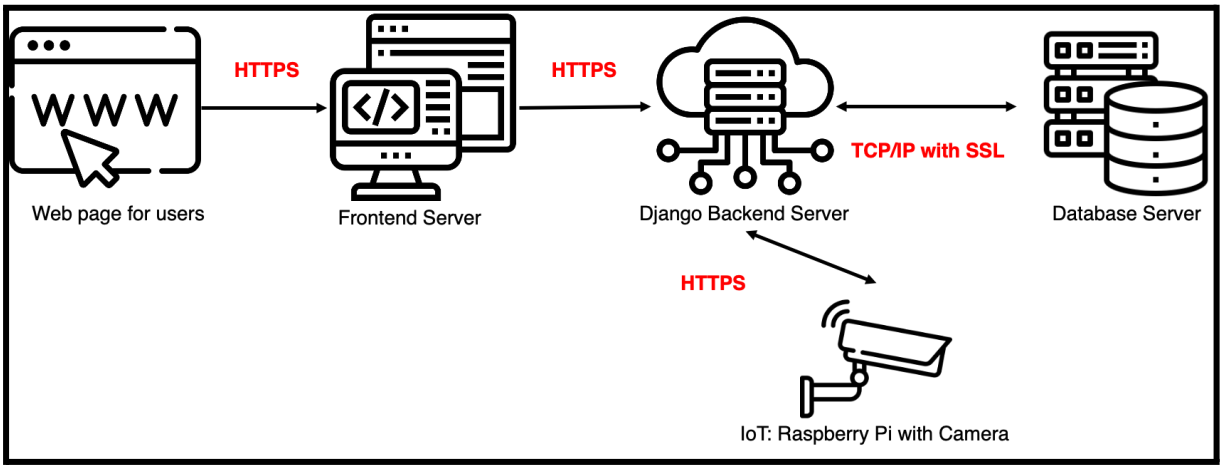


Figure 1: Architecture

2. Roles

This section highlights all the roles, privileges involved in our web application. Our login page allows users to specify their role: Patients, Doctors, Researchers or Medical Staff. After specifying their role, they will be redirected to a portal based on their role after successful authentication.

Patients

Patients are the primary target group of our application and hence, patients can access several functions. The patient's portal enables patients to generate an examination ID, view their health records and view past sessions records. Real-time data from the crowd-sensing IoT devices will be displayed on the patient's portal so that patients can visit the medical facility during less crowded timings.

Doctors

Doctors are the secondary target group of our application. The doctor's portal enables doctors to submit examination ID, view all their past sessions examined, and can examine mode (prescribe and diagnose a patient and submit the details). In addition, Doctors have the ability to view crowd data. More details on Doctor and Patient Subsystem will be covered extensively in Section 7.

Researchers

Researchers portal will allow researchers to view k-anonymised health records. Researchers will be able to define their filter and generate the k-anonymised health records. In addition, Researchers have the ability to download the anonymised health records as a CSV file.

Medical Staff

Medical Staff refer to counter staff that check in the patients to the medical facility. They will be only allowed to view real-time data from the crowd-sensing IoT device. This is to ensure efficient crowd control of the medical facility.

3. Frontend

Overview

The frontend subsystem uses React, a frontend framework written for JavaScript. React uses reusable UI components that can be displayed to the user. It communicates with the back-end via REST APIs as covered in Section 5. The purpose of the frontend is to allow users to view and interact with the application as well as send user data to the backend server to be processed.

3.1 Frontend Technologies

Programming Language & Frameworks:

- React 18.2.0: Framework for building user interfaces based on UI components.
- HTML: Standard markup language displayed in the web browser.
- CSS: Styles sheet language used to style HTML components.

Additional Libraries & Tools:

- Axios 0.27.2: Promise-based HTTP client to send request to server side
- Node.js and npm: Local development platform and package manager
- react-router-dom 6.4.0: Routing in react
- Material UI and related libraries: CSS UI framework that provides simple and customizable React components
- ESLint: Static code analysis tool for JavaScript code

3.2 Frontend Deployment

3.2.1 Setup

The frontend web server is Apache version 2.4.41 running on Ubuntu 20.04.5 LTS. Setup involved installation of Apache and setting up of a virtual host for the frontend app's domain (ifs4205-group3-webapp.comp.nus.edu.sg).

3.2.2 Deploying React App to Apache Server

A .htaccess file was created in the frontend app's directory (/frontend/public/) and enabled in the system's Apache configuration file so that React's BrowserRouter can handle redirection properly via the app's index.html. **`npm run build`** was then used to create a production build of the frontend app in the form of a /frontend/build/ directory. This directory was then copied into a directory that Apache uses to serve documents from.

3.2.2 Running the Server

The command **`systemctl start apache2`** can be used to run the server.

3.3 Frontend Code Structure

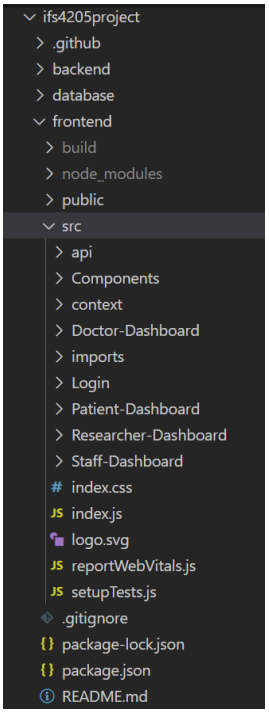


Figure 2: Code structure as seen in VS Code

The code for all UI pages has been organised into 5 main features: Doctor-Dashboard, Login, Patient-Dashboard, Researcher-Dashboard and Staff-Dashboard.

Figure 3a: src/index.js

Figure 3b: src/Login/Login_Home.js

The entry point into the frontend code is index.js which specifies the routes and overall structure of the web app UI. The homepage is the login page (index.js line 37 in Figure 3a), which renders the component returned by the LoginHome() function component exported by Login_Home.js. As seen from line 55 of Login_Home.js (Figure 3b), the function component in turn renders the Login component, which is imported from Login.js (Figure 4). Login.js returns the HTML and CSS for the actual login page seen, makes the API call to the corresponding backend endpoint using data entered by the user into the login form, and handles the response.

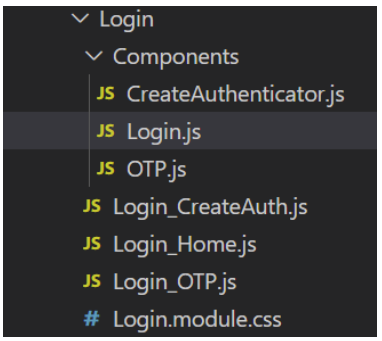


Figure 4: src/Login/Components/Login.js

This is essentially the structure for each of the 5 main features of the frontend code. The root of each feature's folder contains files that export function components used by index.js and import function components in the Components sub-folders. Each file at the root of the folder is responsible for a page that a user of that type can access such as the main page of that feature (_Home.js files). For example, from Figure 4, Login_OTP.js renders the 2FA page that is redirected to after successful login (Figure 20).

3.4 Frontend Security Implementation

3.4.1 AuthProvider

AuthProvider is our authentication provider class. The purpose of an authentication provider is to ensure users have valid credentials before accessing UI resources. In the scenario that an unauthorised user can access the UI pages, they will still not be able to access the backend resources due to the need for an accessToken to authenticate a user (Section 6).

```
/* This Function checks that user is authenticated to access role page*/
export function RequireAuth({ children, role }) {
  const accessToken = sessionStorage.getItem("accessToken");
  const userRole = sessionStorage.getItem("userRole");
  const isVerified = sessionStorage.getItem("isVerified");

  if (accessToken && isVerified && userRole === role) {
    return children;
  } else {
    return <Navigate to="/login" replace />;
  }
}

/* This Function checks that user is init authenticated to access verify otp and create auth page*/
export function RequireInitAuth({ children }) {
  const accessToken = sessionStorage.getItem("accessToken");

  if (accessToken) {
    return children;
  } else {
    return <Navigate to="/login" replace />;
  }
}

/* This Function checks that the user is authenticated to access Examination Page */
export function RequireExam({ children }) {
  const examId = sessionStorage.getItem("examId");
  const patientName = sessionStorage.getItem("patientName");
  const userRole = sessionStorage.getItem("userRole");

  if (examId && patientName && userRole === "Doctor") {
    return children;
  } else {
    return <Navigate to="/doctor" replace />;
  }
}
```

Figure 5: src/context/AuthProvider.js

In our AuthProvider class, we have several functions such as RequireAuth, RequireInitAuth and RequireExam which are exported and utilised in our Index.js. These functions as seen in Figure 5 define the authentication conditions required to access the page. If users do not meet the requirements, this means they are not authenticated and they will be redirected to their root page in an attempt to access the page.

```
<Route
  path="healthrecords"
  element={
    <RequireAuth role="Patient">
      <PatientRecords />
    </RequireAuth>
  }
/>
<Route
  path="doctor"
  element={
    <RequireAuth role="Doctor">
      <DoctorHome />
    </RequireAuth>
  }
/>
```

Figure 6: Index.js using the AuthProvider functions

Figure 6 shows how the authentication functions from the AuthProvider class are utilised. Index.js defines all the routes and components of our web application. We wrap each user interface component with the required authentication class. For example, to access a doctor page (/doctor), the patient must be actually a doctor to access this page (Figure 6). If an unauthorised user tries to access the page, they will be redirected to /login as defined by the RequireAuth function in AuthProvider (Figure 5).

3.4.2 HTTPS

Our frontend server contains a SSL certificate which enables HTTPS on our frontend server. As such, this prevents our website from having information broadcasted in a way that can be easily viewed by a man-in-the-middle snooping on the network.

4. Backend

Overview

The backend subsystem uses Django, a server side web framework written in Python. It communicates with the front-end via REST APIs and handles communication with the database. Django handles the application logic and implements a significant portion of the application's security including authentication and permissions.

4.1 Backend Technologies

Programming Language & Frameworks:

- Python 3.9
- Django Web Framework 4.1.2

Additional Libraries:

- Django REST framework 3.13.1
- Django-otp 1.1.3
- Django-filter 22.1
- Django-cors-headers 3.13.0
- Psycopg2 2.9.3 (PostgresQL database adapter for Python)

4.2 Backend Deployment

The backend web server is Apache version 2.4.41 running on Ubuntu 20.04.5 LTS. Setup involved installation of Apache and setting up of a virtual host for the backend app's domain (ifs4205-group3-backend.comp.nus.edu.sg). Apache accepts HTTPS connections and HTTP connections are redirected to use HTTPS.

4.3 Backend Security Implementation

4.3.1 Authentication

Django's REST framework library provides authentication schemes which we use in order to authenticate a user. We implemented a token-based authentication scheme and two-factor authentication (2FA) using Google authenticator along with Django's django-otp library. Section 6 describes the authentication process in more detail.

4.3.2 Permissions (Role-based access control)

Our web application's functionality is different for each user role, and each user should not be able to use a functionality not meant for their role. Hence, we use Django's built-in permissions system, customised to our project's needs to implement role-based access control. Django allows us to assign permissions to specific groups of users. By requiring users to choose their role before they can login, we are able to assign users to a specific group based on their role.

```python
class IsPatient(permissions.BasePermission):
    message = "This resource can only be accessed by patients."

    def has_permission(self, request, view):
        try:
            Patient.objects.get(user=request.auth.user)
            return True
        except Patient.DoesNotExist:
            return False
```

Figure 7: Sample source code of patient permission class

For example, when a patient tries to access a web page meant for patients, we check if the user is in our database of patients. If the user appears in our patient database, he will receive permissions to view the specified web page. Every web page (except for the login page) requires the user to be authenticated and have the **required permissions** in order to have access to the web page. Using this method, users can then only access the web pages which are meant for their role (e.g. patient viewing health records, researcher viewing anonymised data).

4.3.3 Cross Site Request Forgery (CSRF) protection

```
GET /doctorviewrecords HTTP/1.1
Accept: application/json, text/plain, */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,ko;q=0.8
Authorization: Token 248b40b0a2cd601c059b54c0e96ace7c03f50db1
```

Figure 8: Example of GET request header

CSRF attacks allow a malicious user to execute actions on behalf of another user without them knowing. Through the use of authorization tokens included in the HTTP requests sent to the backend, CSRF attacks are avoided provided the attackers do not know the value of the token. As long as the authorisation field in the HTTP header is missing or does not have a correct token, the request is dropped by the backend. The tokens are stored in our database which is inaccessible to users outside the SoC network and are only included in the HTTPS requests, which are encrypted.

4.4 Backend Testing

We use unit tests to ensure that our APIs are working as intended. These tests are integrated in our DevOps pipeline and are run on every push and commit.

```python
class PatientLoginTest(TestCase):
    def setUp(self):
        self.user = get_user_model().objects.create_user(
            username="patient", password="12patient12", email="test@example.com"
        )
        self.user.save()
        Patient.objects.create(user_id=self.user.user_id)

    def tearDown(self):
        Patient.objects.filter(user_id=self.user.user_id).delete()
        self.user.delete()

    def test_login_pass(self):
        response = self.client.post(
            "/login",
            {"username": "patient", "password": "12patient12", "role": "patient"},
        )
        expected_response = status.HTTP_200_OK
        self.assertEqual(response.status_code, expected_response)

    def test_wrong_username(self):
        response = self.client.post(
            "/login",
            {"username": "wrong", "password": "12patient12", "role": "patient"},
        )
        expected_response = status.HTTP_403_FORBIDDEN
        self.assertEqual(response.status_code, expected_response)
```

Figure 9: Sample source code of our Patient login tests

Above is an example of a few test cases for our login API. We create test cases for the most common user actions. For example, for our login API tests, we have tests for correct credentials as well as incorrect credentials or unauthorised roles.

4.5 Backend Logging

```
[INFO] [2022-10-21 21:14:52,589] User | doctor_alicia | /login | Successful
[INFO] [2022-10-21 21:14:57,881] OTP | doctor_alicia | /verifyotp | Success | Token verified
[INFO] [2022-10-21 21:14:58,102] Doctor | doctor_alicia | /doctorviewoldsessions | Success
[INFO] [2022-10-21 21:14:58,152] Doctor | doctor_alicia | /doctorviewoldsessions | Success
[INFO] [2022-10-21 21:15:11,068] User | patient_alicia | /login | Successful
[INFO] [2022-10-21 21:15:17,210] OTP | patient_alicia | /verifyotp | Success | Token verified
[INFO] [2022-10-21 21:15:33,386] Doctor | doctor_alicia | /assigndoctor | Success | exam_id = eSpcvRahUZ
```

Figure 10: Logs generated by backend

We have implemented logging for all of our APIs. Logs contain information such as the API name, the user calling the API, whether it is successful or a failure as well as any additional information from the API. The logs are useful to track and investigate any suspicious activity performed by a particular user.

5. APIs between Components

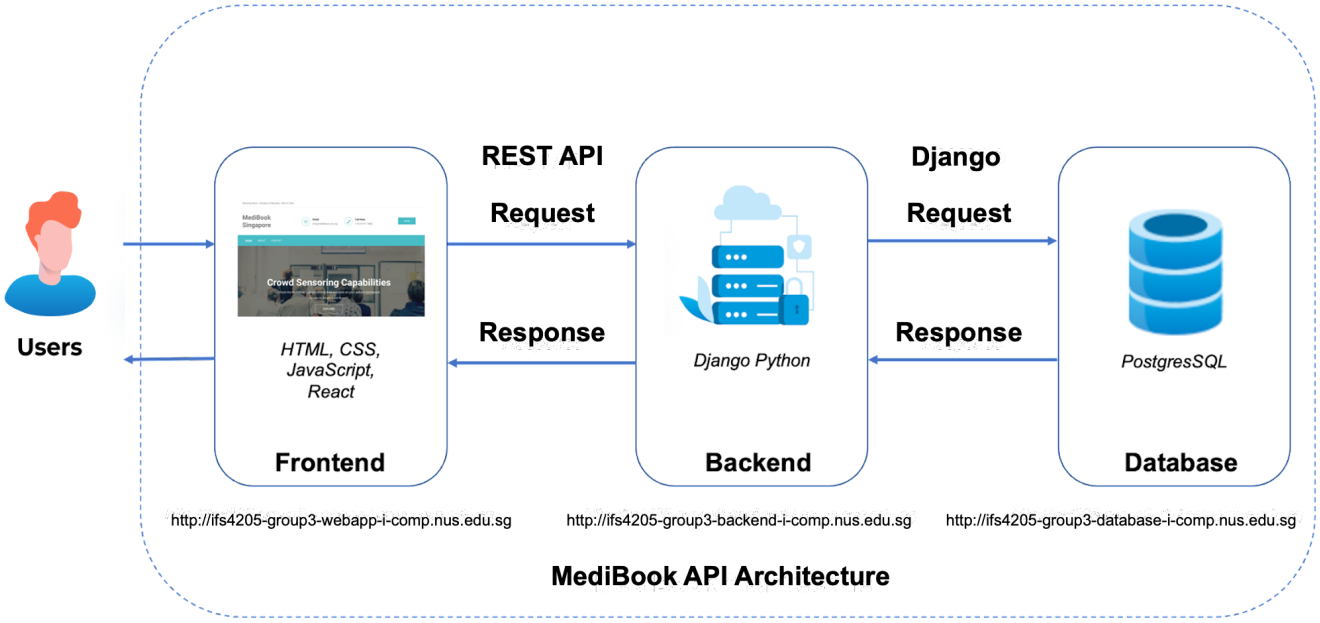


Figure 11: API Architecture

This section will detail the APIs used to send request and response from 1) frontend and backend and 2) backend and database. The main logic of each component is that it will send a request and receive a response. All three of our components are hosted on Ubuntu Virtual Machines with the frontend hosted on http://ifs4205-group3-webapp-i-comp.nus.edu.sg, backend server hosted on http://ifs4205-group3-backend-i-comp.nus.edu.sg and our database server hosted on http://ifs4205-group3-database-i-comp.nus.edu.sg.

5.1 Frontend and Backend

Frontend and Backend use REST API as its application programming interface to allow interaction between our frontend and backend web services. REST stands for Representational State Transfer and it is a set of architectural constraints that can be designed in several ways.

For our web application, when a frontend request is made via a RESTful API, it transfers a representation of the state of the resource to the backend. This information is delivered via HTTP using JSON (JavaScript Object Notation). In addition, our web application uses headers and parameters as part of our HTTP methods of a RESTful API HTTP request as we include the access token to ensure the user is authorised to access the resources. This is covered in Section 6.

In addition, our web application strictly uses two different HTTP request types: GET and POST. We use POST requests when we want to send data such as form data to the server. On successful creation, the backend returns an HTTP status code of 200 with a JSON response that will be parsed and displayed on the frontend. For example, when a user submits login details, our frontend prepares a JSON with the form data and sends it to the

backend. The backend will send back a JSON response with an access token as well as our details for the frontend to process to the next page via the user interface/user experience.

The web application also uses GET requests when the user from the frontend wants to read/retrieve data from the backend, successful GET requests will return a JSON response with HTTP status code 200. The frontend will process the JSON response and display it to the user via user interface/ user experience. For example, the GET request is used when our user dashboard is rendered to the user. As part of the user interface, the crowd data will be shown to the user. Every 1 minute, the frontend sends a GET request to the backend endpoint to retrieve the crowd data and display it to the user.

```
useEffect(() => {
  axios
    .get(LOGIN_AUTH_URL, {
      headers: {
        "Content-Type": "application/json",
        Authorization: tokenString,
      },
    })
    .then(function(response) {
      setQrString(response.data.message);
      setSuccess(true);
      setBuffer(false);
    })
    .catch(function(err) {
      setFailure(true);
      setBuffer(false);
      if (!err.response) {
        setErrMsg("No Server Response");
      } else if (err.response.status === 400) {
        setErrMsg(err.response.data.message);
      } else if (err.response.status === 401) {
        setErrMsg(err.response.data.message);
      } else if (err.response.status === 403) {
        setErrMsg(err.response.data.message);
      } else if (err.response.status === 405) {
        setErrMsg(err.response.data.message);
      } else if (err.response.status === 500) {
        setErrMsg(err.response.data.message);
      } else {
        setErrMsg("Server encountered an error, please try again.");
      }
    });
}, [tokenString]);
```

Figure 12: Sample Source Code in Frontend using axios

Our frontend uses Axios library to process all the HTTP requests in the frontend. Axios is an HTTP client library based on promises. It allows us to send asynchronous HTTP requests to REST endpoints. Figure 12 shows sample source code in the frontend where we are trying to generate 2FA QR code from the frontend to the backend endpoint. We send a GET request to **`LOGIN_AUTH_URL`** which is the endpoint we are trying to reach with the access token in the headers. On successful response with a HTTP status code 200, the backend returns a response which is parsed by the frontend. In the case of an error, axios will catch the error and allow us to process the error and display an error message based on the response message from the backend.

Login

Description	This API authenticates user
Endpoint	http://ifs4205-group3-backend-i.comp.nus.edu.sg/login
HTTP Method	POST

Sample Request	`{"username":"username", "password":"abcd", "role":"patient"}`
200 Response	`{"token":"token", “Name”: “John”, “role” : “patient”, "hasDevice": {true/false} }`
405 Response (i changed this from 400 to 405 as it will make handling the exception much easier on the backend)	`{“message”: “Invalid request method.”}`
403 Response	`{“message”: “Login credentials are incorrect. Please check and try again.”}`

Figure 13: Login API Definition

We group our APIs based on functionality, namely:

- Login/logout API (authentication)
- Patient APIs
- Doctor APIs
- Researcher APIs
- OTP APIs
- IOT APIs

To facilitate easier development, our team created an API document to define each API call that is used in our web application. Figure 13 shows the API definition of the Login API. In each API call, we define the description, backend endpoint that will process the request and the HTTP method (GET, POST). In addition, we define a sample correct request and a sample 200 response. We also define 405 and 403 responses so that the frontend knows what to expect when there is an error.

With a proper API document, the frontend developers can create a request based on the parameters defined by the API document and write code to expect errors based on the API document as well. In addition, the backend developers can write code expecting a request as defined by the parameters of the API document. This ensures that frontend and backend

can work independently to implement the APIs without the constant need to integrate the frontend and backend. The full list of our APIs can be found in our API Document.

On the backend, we use Django's REST framework library to parse JSON requests from the frontend, as well as reply with JSON responses.

```python
class PatientViewRecords(APIView):
    parser_classes = [JSONParser]
    permission_classes = [IsVerified, IsNotExpired, IsPatient]

    def get(self, request):
        user_obj = request.auth.user
        patient_obj = get_patient_object(user_obj)
        data = {}
        # Checks if user is a patient
        if not patient_obj:
            log_info(
                [
                    "Patient",
                    user_obj.user.username,
                    "/patientviewrecords",
                    "Failure",
                    "Unauthorised",
                ]
            )
            return Response(
                {"message": "Action forbidden."}, status=status.HTTP_403_FORBIDDEN
            )
        try:
            record_obj = HealthRecord.objects.get(pk=patient_obj)
            data["healthRecords"] = PatientRecordsSerializer(record_obj).data
        except HealthRecord.DoesNotExist:
            data["healthRecords"] = {}
        try:
            past_sessions = Examination.objects.filter(patient=patient_obj)
            data["examRecords"] = PatientPastSessionSerializer(
                past_sessions, many=True
            ).data
        except Examination.DoesNotExist:
            data["examRecords"] = {}
            log_info(
                ["Patient", request.user.username, "/patientviewrecords", "Success"]
            )
        return Response(data, status=status.HTTP_200_OK)
```

Figure 14: Sample source code of patient API

Above is an example of one of our patient APIs. Our APIs follow a standard structure: first, we check whether the user is permitted to access the API through the permission classes (highlighted in red box). Then, we parse the JSON request and the message (if any) and execute the application logic. If necessary, we query data from our database using Django's data abstraction API to retrieve our data in the form of Django objects (highlighted in green box). Logging is done whenever exceptions occur or when the API executes successfully (highlighted in blue box).

5.2 Backend and Database

Django provides a database abstraction API that allows us to create, retrieve, update and delete objects. Django objects are instances of models which reflect data stored in a database. Each model generally maps to a single database and attributes of a model represent a database field. In the example below, the *HealthRecord* model represents the *HealthRecord* table in our database. Its attributes like height, weight and allergies represent the corresponding fields in the table.

```
class HealthRecord(models.Model):
    user = models.OneToOneField(Patient, on_delete=models.CASCADE, primary_key=True)
    dateofbirth = models.DateField()
    sex = models.CharField(max_length=2)
    height = models.DecimalField(max_digits=5, decimal_places=1)
    weight = models.DecimalField(max_digits=5, decimal_places=1)
    bloodtype = models.CharField(max_length=3)
    allergies = models.CharField(max_length=50)
    race = models.CharField(max_length=10)
    zipcode = models.CharField(max_length=6)
    address = models.CharField(max_length=100)
```

Figure 15: Sample Source code of HealthRecord model

When querying for objects, Django's database abstraction API uses prepared SQL statements and are constructed using query parameterisation, which mitigate SQL injections.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'HOST': get_secret('HOST'),
        'NAME': get_secret('NAME'),
        'USER': get_secret('USER'),
        'PASSWORD': get_secret('PASSWORD'),
        'OPTIONS': {
            'sslmode':'require',
        },
    }
}
```

Figure 16: Sample source code of the settings file in Django

To further enhance security, we use SSL connections to encrypt communication between the backend and the database. Due to the sensitive nature of the information being transferred between the backend and the database, it is paramount that information is encrypted such that no malicious actor is able to eavesdrop on the queries sent to the database.

6. Authentication

Assumptions

- Users can only create an account by being at the facility in-person
- Users have a secure device that can be used for 2FA
- The email account that was used when registering for an account is valid and uncompromised

6.1 Login subsystem

Overview

The login system is used as the method to authenticate users in the system. All users must login before they can begin using any resources. There is no registration function on the application. Users are assumed to be given an account when they physically visit the facility for the first time.

6.1.1 Implementation

When a user tries to login through the frontend, the frontend will send a POST request to the backend server containing the username, password and role that the user is trying to login as.

The backend will verify the credentials received. If the credentials are incorrect or the user attempts to sign in as a role that he or she is not assigned to, a generic error message is returned to prevent information leakage. Upon successful verification, it responds to the frontend with a token. The frontend then stores the token, name and role of the user in sessionStorage. This read-only JavaScript property is cleared when the tab or browser closes and the page session ends.

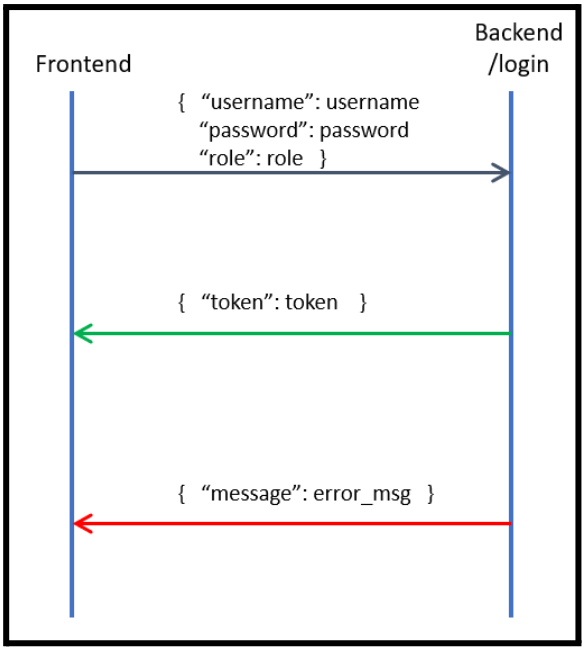


Figure 17: Login Authentication Module Diagram

6.1.2 Token

The token is a 256-bit hex string generated using Python ***secrets*** library. A new token is generated for every user login. Each user can have at most one token at any given time, hence each token is able to **uniquely identify** a user. If a user already has an existing token, the token value is overridden rendering the previous token unusable. The token also contains a *verified* field which is used by the two-factor authentication subsystem and a *role* field which tracks which role did the user sign in as.

If a user is both a patient and a doctor, signing in as a doctor will grant the user permission to access doctor APIs only. To access patient APIs, the user must relog as a patient.

All requests to the backend API must contain the token as part of the request header in the **Authorization** field. If a request does not contain the token or contains an invalid token, the backend will reply with '401 Unauthorised' response.

Each token expires after **24 hours**. The database stores the timestamp of each token creation. Upon every request, this timestamp is checked. Users must login again if the token has expired.

6.2 Two factor authentication subsystem

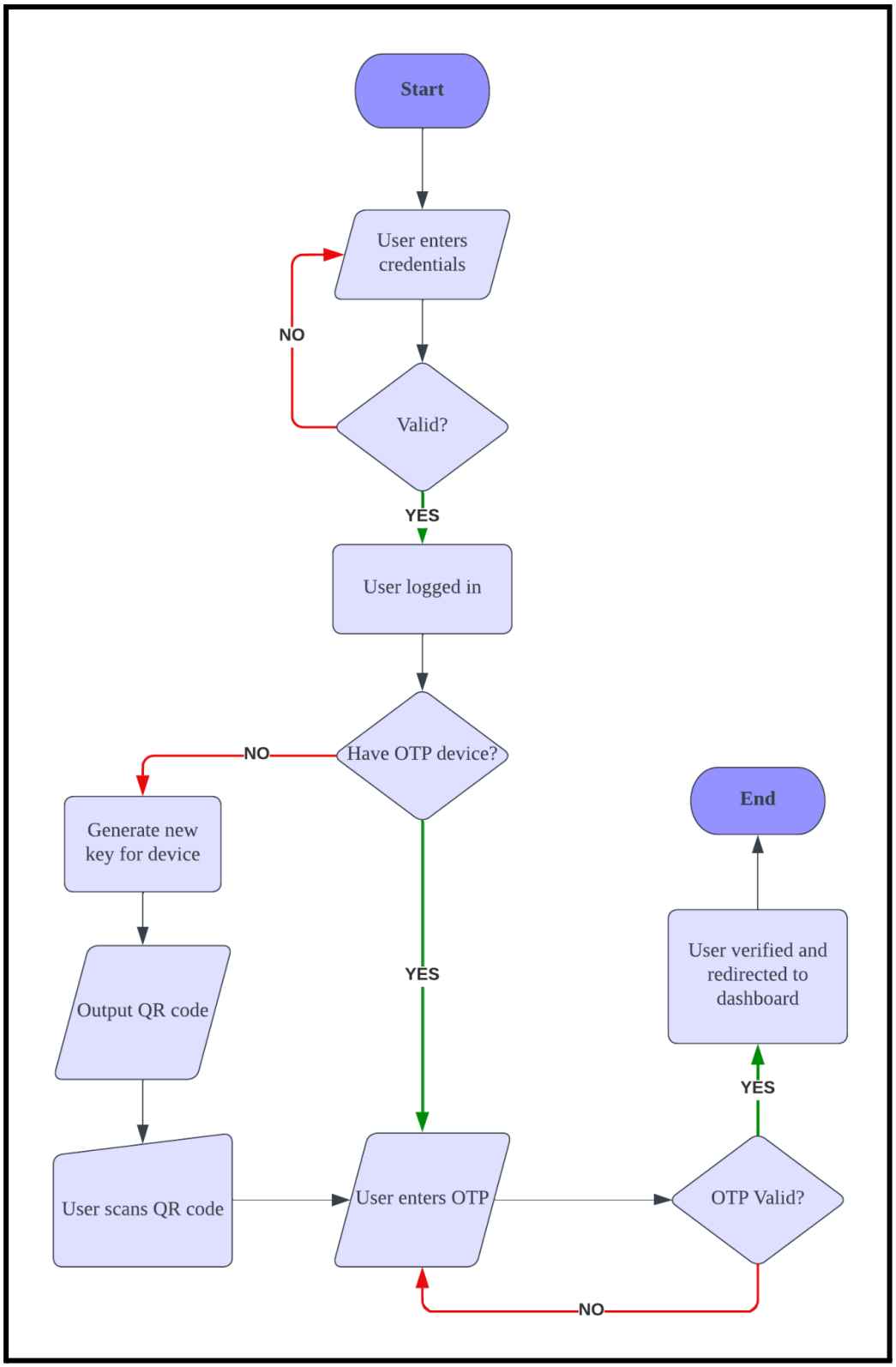


Figure 18: Login Authentication Flow

Overview

To further strengthen security, users must register a Time-based One-time Password (TOTP) device that can generate an OTP for verification. All backend API (excluding login and logout) requires OTP verification before use.

6.2.1 Implementation

The ***django-otp*** plugin was used to implement the 2FA subsystem. If the user does not currently have any ***confirmed*** device registered, the user will be prompted to register a new device.

When the backend receives a request from a user to register a new device, the backend will respond with a TOTP key that looks something like:
'*otpauth://totp/Example:alice@google.com?secret=JBSWY3DPEHPK3PXP&issuer=Example*'

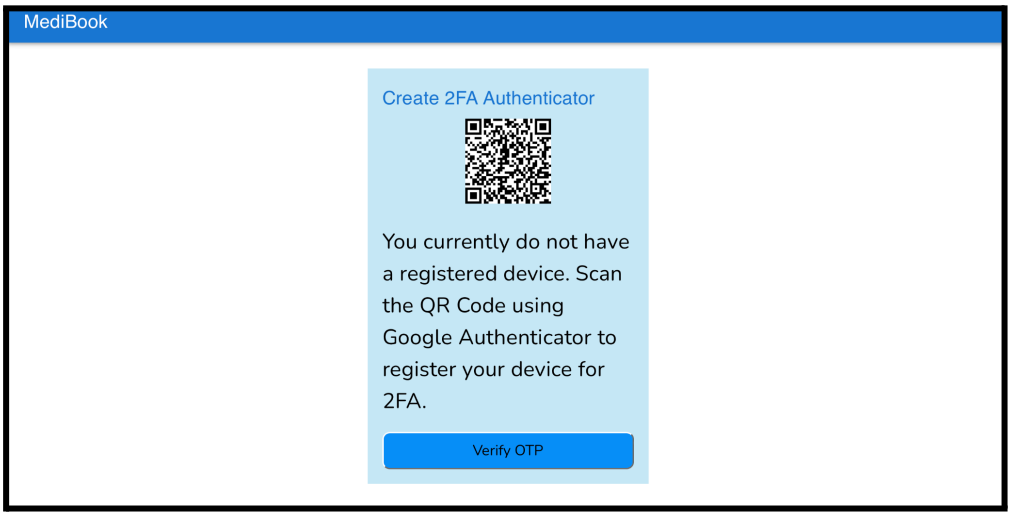


Figure 19: Create Authenticator Page

The frontend will generate a QR code using the key above for the user to scan with an authenticator app such as Google Authenticator as seen from Figure 19. Once the authenticator is linked with the account, the user will click *Verify OTP* as seen from Figure 20 and can get the OTP from their authenticator as seen from Figure 21 and submit the OTP.

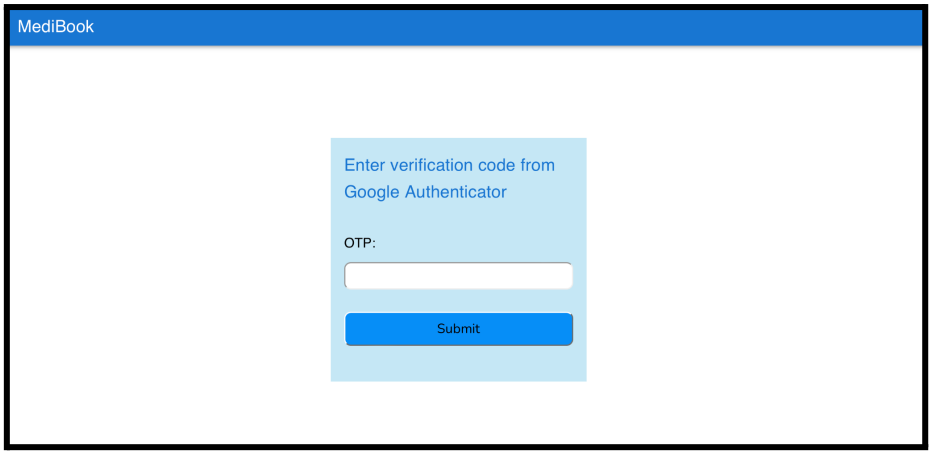


Figure 20: Verify OTP Page

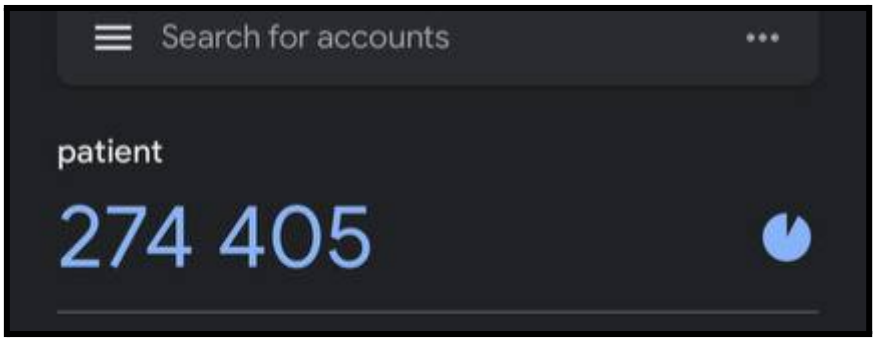


Figure 21: Google Authenticator Page

If the OTP was successfully verified by the backend, the token for that user is considered ***verified***. When a user successfully verifies an OTP for the first time, the device is set as ***confirmed***. All requests to the backend excluding login and logout require a ***verified*** token.

6.2.2 Changing Authenticator:

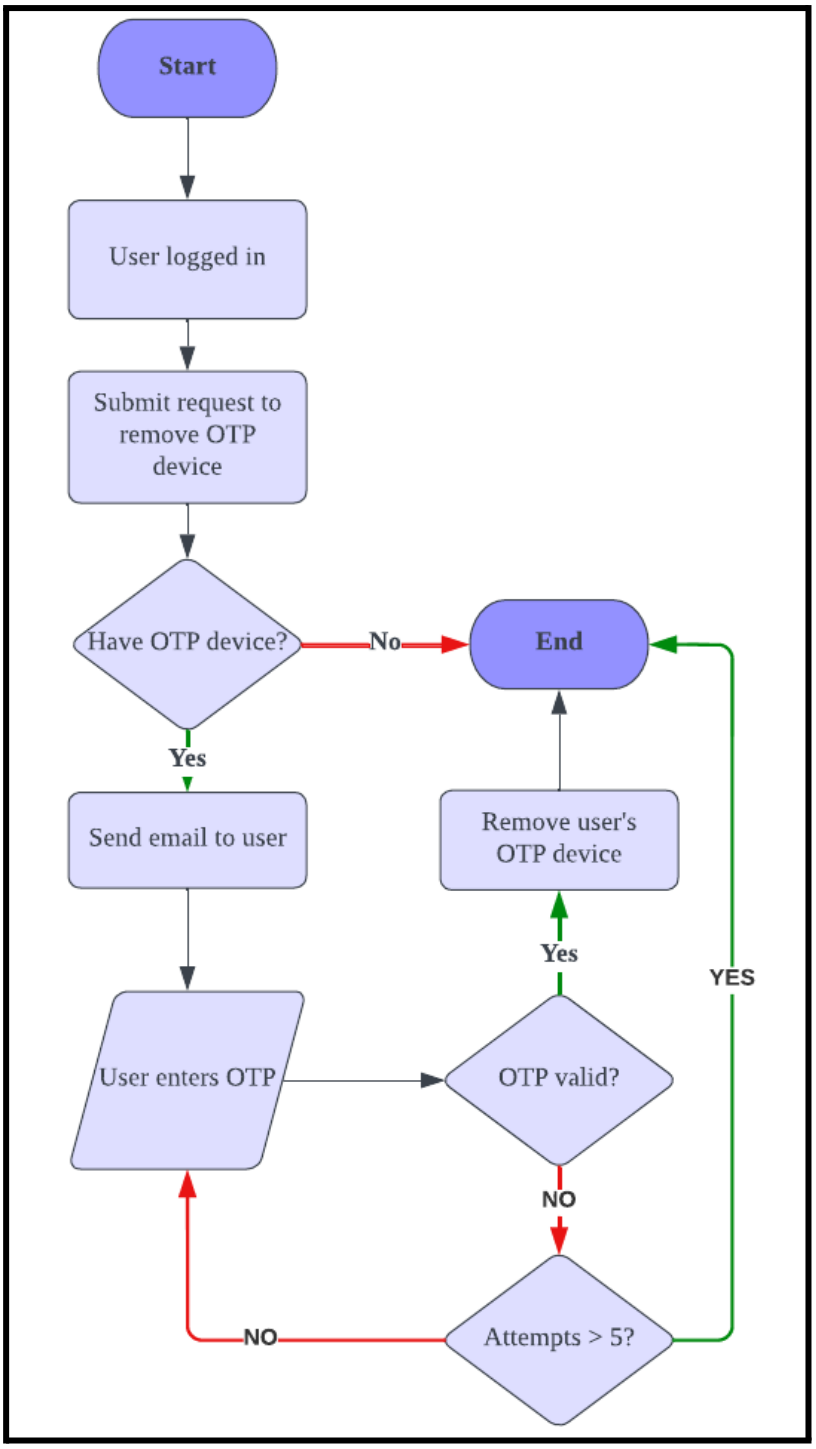


Figure 22: Changing Authenticator Flow

In the event where a user wants or needs to change their authenticator device, they can go to the settings page to change their authenticator.

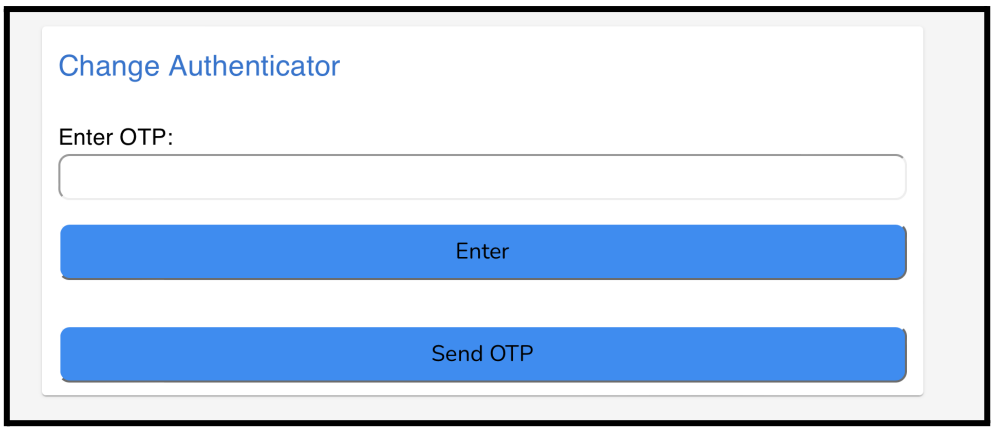


Figure 23: Google Authenticator Page

To change the authenticator, the user has to click on the *Send OTP* button which will prompt the backend to send an email containing an OTP to the email address associated with the account. A sample email can be seen from Figure 23. If the user enters the wrong OTP more than 5 times, the OTP is invalidated and a new OTP must be requested.

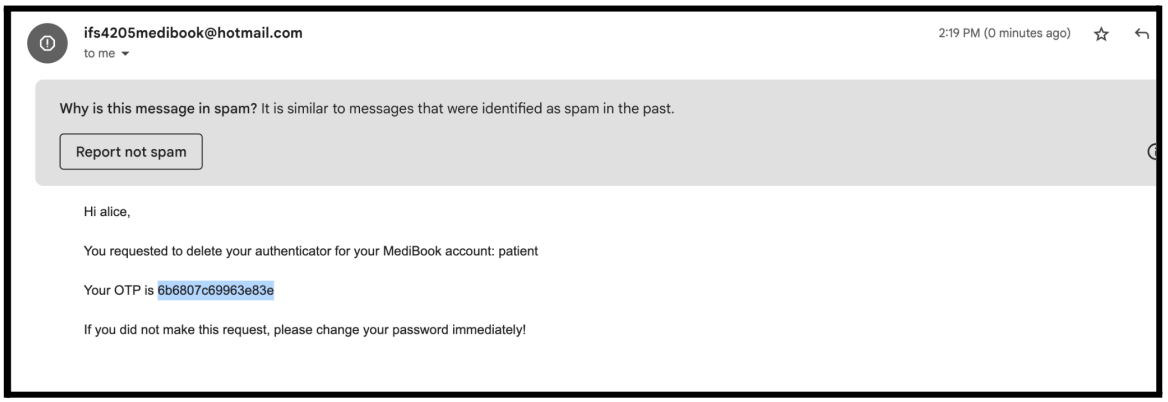


Figure 24: Change Authenticator Sample Email with OTP

Upon getting the OTP from the user's associated email, they can submit the OTP back at the Settings page. Successful submission of the correct OTP will successfully remove the device. The user will then have to relogin and set a new authenticator again.

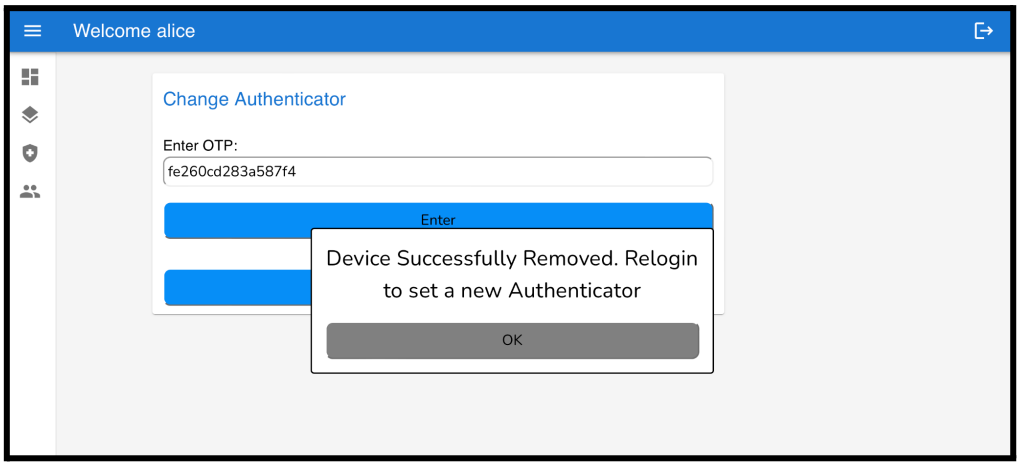


Figure 25: Successful device removed

7. Doctor Patient Flow Subsystem

Overview

This subsystem explains the doctor and patient flow subsystem. When a patient sees a doctor, our web application defines their medical checkup as an examination. Functions that are accessible by patients and doctors are stated here, patients are allowed to generate an examination ID and view their health records and examination records. Doctors can examine patients and record details of the current visit such as diagnosis and prescriptions (if any), they can also view patient's health records for reference during their examination session only after given explicit consent by patients which is enabled on the patients' portal

7.1 Flow Process

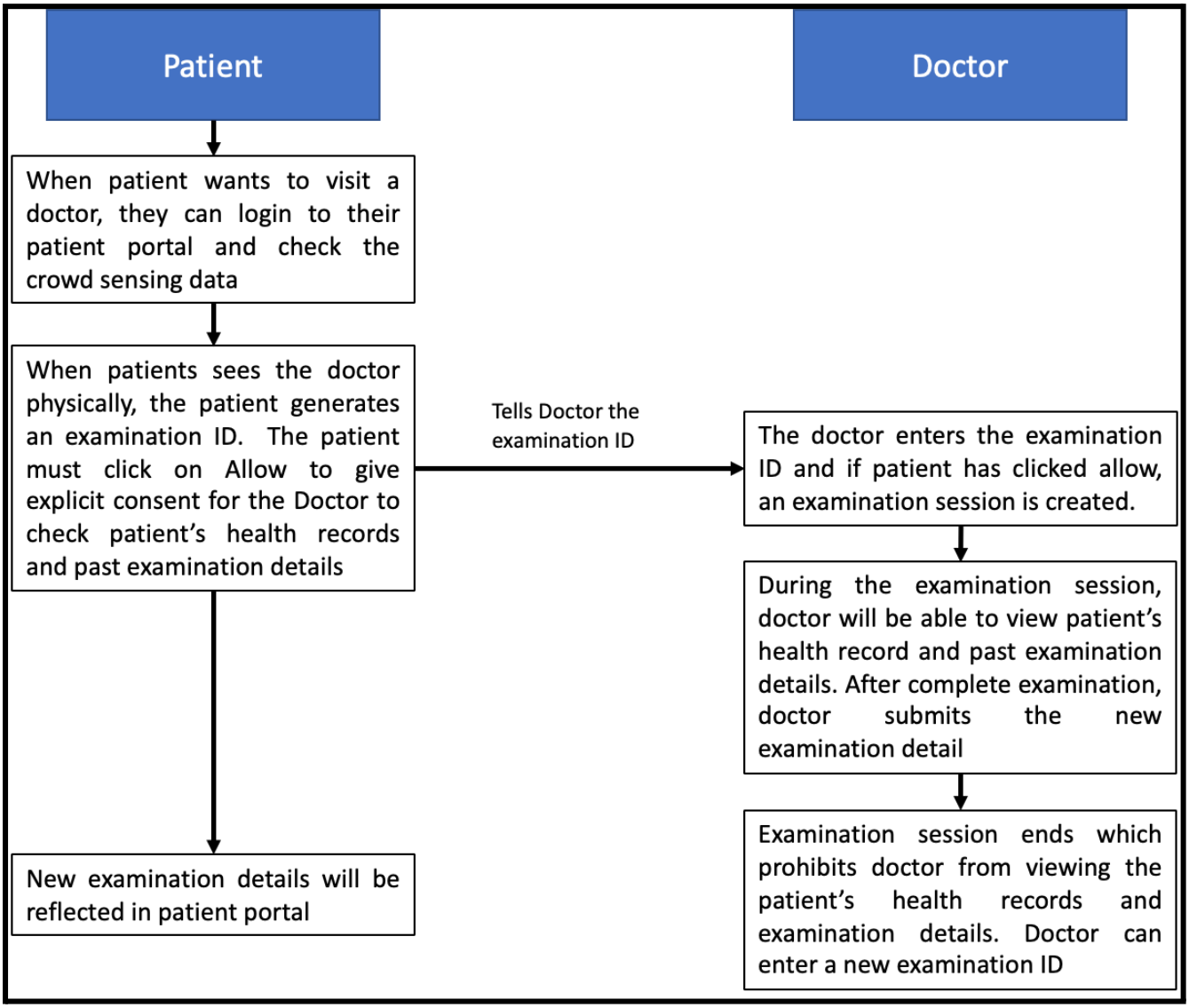


Figure 26: Patient Doctor Flow

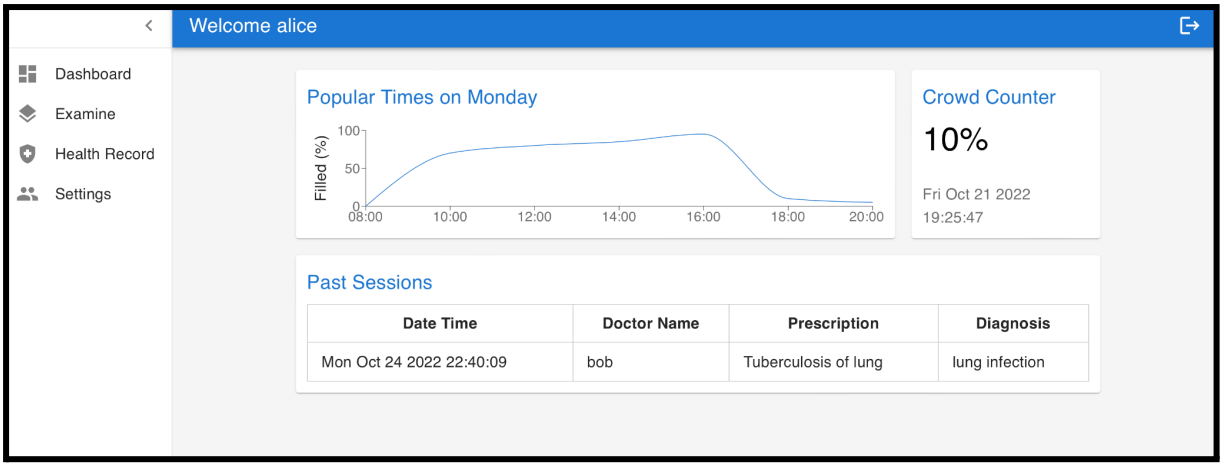


Figure 27: Patient Dashboard

When a patient wants to visit a doctor, they can login to their patient portal and check the crowd sensing data. They also can view popular timing on the certain day to get historial crowd data to get a rough sensing of the potential crowd in upcoming hours. They also can see all historical past sessions and view their own personal health record which is created when a user signs up with MediBook. To create an examination ID, they can navigate to *Examine*.

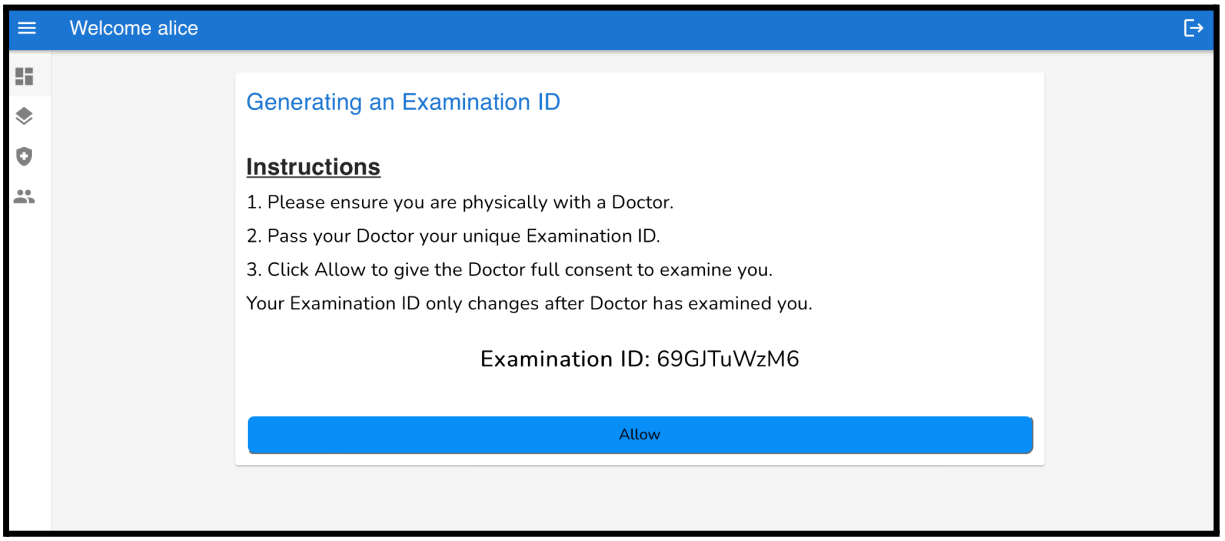


Figure 28: Patient Generate Examination ID Page

When a patient sees the doctor physically, the patient generates an examination ID. The patient must click on Allow to give explicit consent for the Doctor to check the patient's health records and past examination details. The patient then physically informs the doctor of the examination ID which is to be submitted on the Doctor’s portal.

Figure 29: Doctor Enter Examination ID Page

On the doctor's dashboard, when they are with the patient physically, they can access the examination mode by navigating to *Examine*. If there is an ongoing session, the doctor can access the ongoing session form by navigating to *Current Session*.

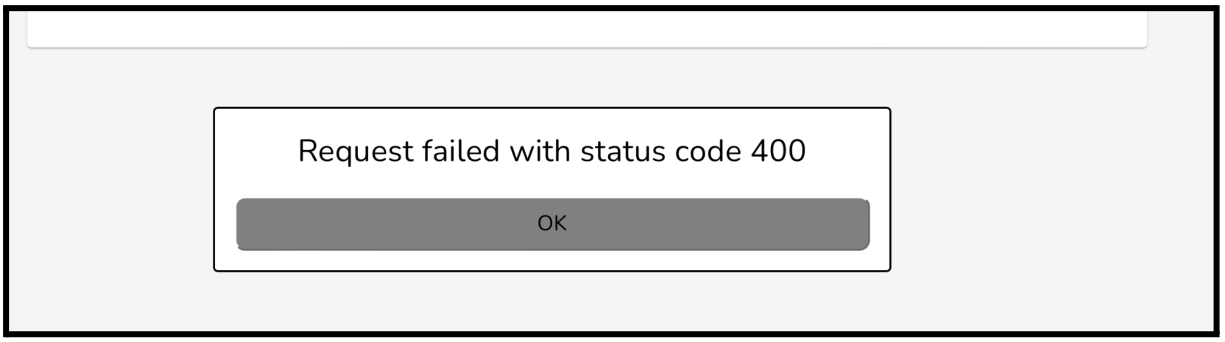


Figure 30: Doctor Enter Examination ID Error

The doctor enters the examination ID and if the patient has clicked allow, an examination session is created. If the patient has not clicked allow on their portal, on entering the session ID, the doctor will receive the error message as seen from Figure 31.

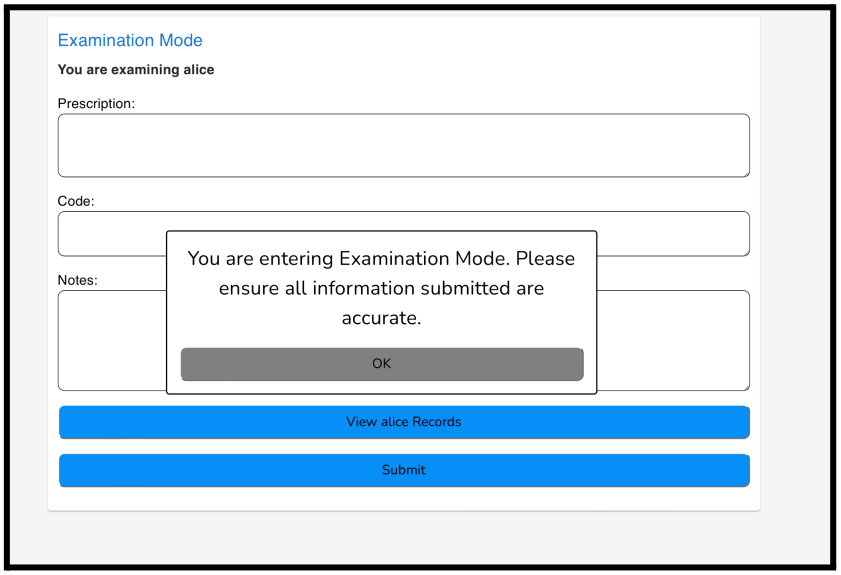


Figure 31: Doctor Examination Mode

After successful submission of a valid examination ID, the doctor enters examination mode which is a confidential mode. During this period, doctors can view their patient's health record and past examination details. The doctor is only allowed to view their patient's health records and past sessions during the examination session. Once the doctor submits the prescription and code for the current examination session, the session ends and the doctor no longer has access privilege to patient details.

Figure 32: Doctor View Patient Records

MediBook also follows the international standardised medical codes from ICD-10. ICD-10-CM stands for International Classification of Diseases, Tenth Revision, Clinical Modification and is a system used by physicians and other healthcare providers to classify and code all diagnoses, symptoms and procedures recorded in conjunction with hospital care in the United States. MediBook uses the ICD-10 as a means to standardise all recorded diagnosis so when a doctor has finished examining the patient, they will have to enter the standardised medical code as defined by ICD-10.

7.2 Security Implementation

1. Doctors have to receive explicit consent from a patient to be able to examine and view patient health records.
    a. This is to ensure sensitive data is protected and only allowed to be accessed after obtaining the patient's consent.
2. Health records cannot be modified after the doctor has submitted examination details for the current session.
    a. This is to ensure the integrity of the examination session and to prevent unauthorised modification and authorisation after an examination session has ended.
3. Patient data can only be viewed during examination sessions.

    a. This is to ensure the integrity of the examination session and to prevent unauthorised modification and authorisation after an examination session has ended.
4. Each user is only allowed to generate one unique examination ID at any given time to prevent spam.
    a. This is to ensure the integrity of the examination session.

8. Researcher k-anonymity Subsystem

Overview

The application has a portal for researchers to access anonymized data for epidemiology purposes. The source of data is the generated data described in **section 10.2**. There are currently over 9000 anonymised records in the application. The records have been processed to have k-anonymity and l-diversity where k = 10 and l = 5.

8.1 Process

The tool used for anonymization is **ARX Anonymization Tool**. It is an open-source software that supports multiple anonymous models such as l-diversity, t-closeness and others.

The software allows hierarchies to be specified for each quasi-identifier which is used for generalisation. Figure 33 shows a snippet of the hierarchy used for generalising age is shown below.

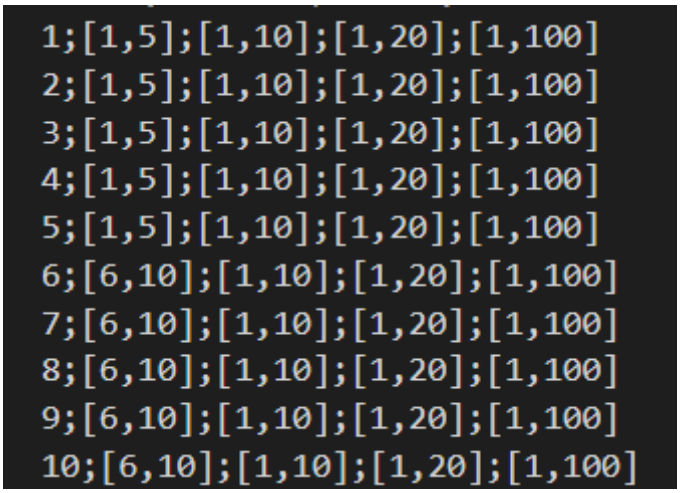


Figure 33: Hierarchy used for Generalising age

In this case, a patient's age can be generalised to different levels going from an exact number to a range between 1-100 years old.

For our dataset, the quasi-identifiers and its range are:

- Age: [6-100]
- Height: [130.0-209.0]
- Weight: [40.0-119.0]
- Zip-code: [100000-900000]
- Allergies: [Eggs | Milk and Dairy | Peanuts | Tree nuts | Fish | Shellfish | Wheat | Soy | Sesame | None]
- Race: [Chinese | Malay | Indian | Others]
- Sex: [M | F]

When extracting the health records and examination data, the identifying attribute patient_id is omitted to prevent re-identification. Although the anonymisation tool has the capability to suppress identifying attributes, it replaces the attribute with a *. This means the exported

data will require additional processing before it can be imported into the database. Hence, we opted to omit patient_id when anonymising the data.

The diagnosis code is the only sensitive attribute. Hence, when querying specific diagnosis code, there is a possibility that less than *k* rows will be returned although this is perfectly acceptable by definition of *k*-anonymity. The diagnosis codes used are from ICD-10 2022 codes.

The anonymised data is exported into a CSV file which is imported into the PostgreSQL server. Refer to Figure 40 (ER diagram) to see the schema of the anonymised table.

8.2 Querying the data

The data can be retrieved by researchers in the researcher portal. A researcher is allowed to query using 1 or more of any quasi-identifiers. For fields like zipcode and age, only specific values are allowed to be used. Using a range of values as input will result in an error message. However, **"*"** can be used on any field if no specific value is needed.

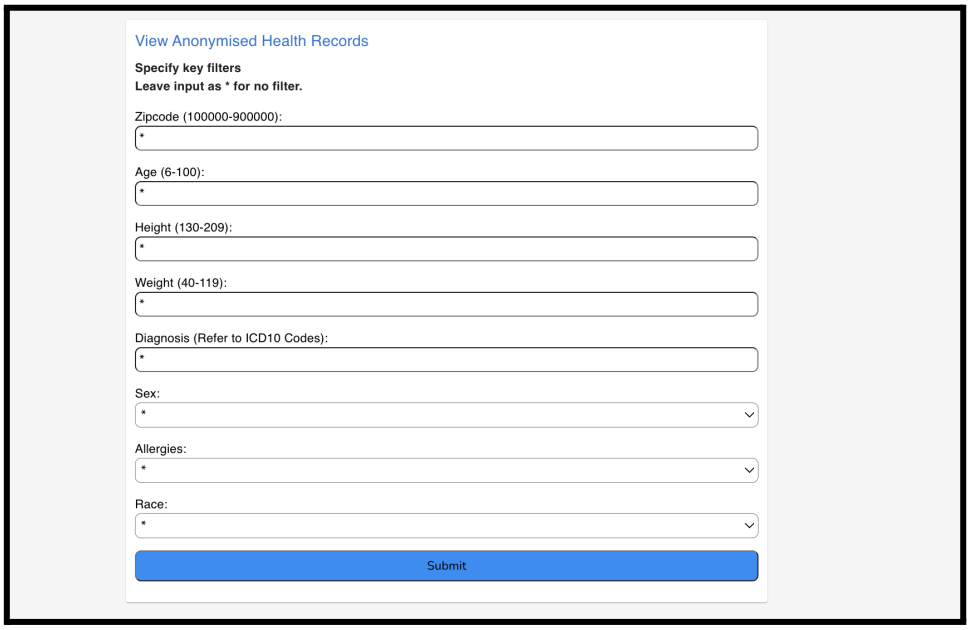
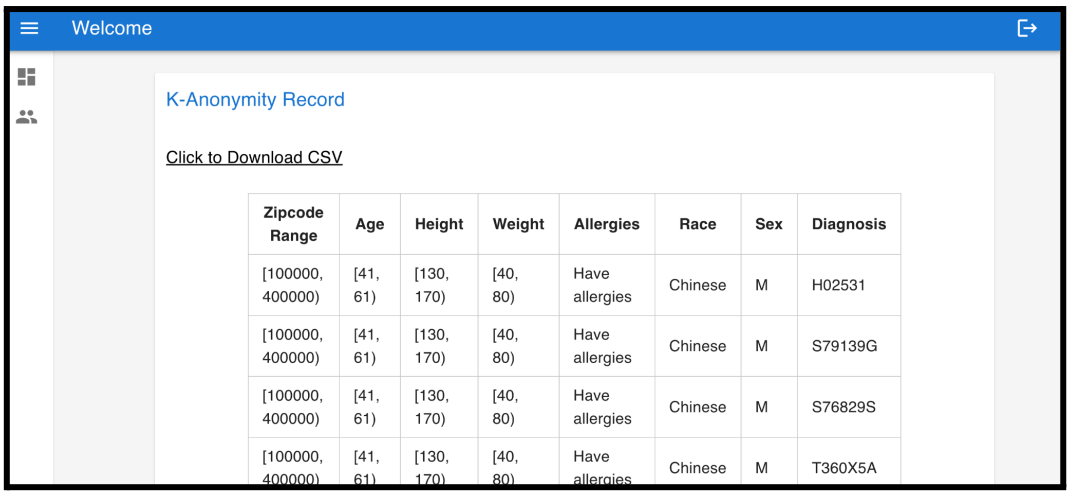


Figure 34: Querying Data



Zipcode Range	Age	Height	Weight	Allergies	Race	Sex	Diagnosis
[100000, 400000)	[41, 61)	[130, 170)	[40, 80)	Have allergies	Chinese	M	H02531
[100000, 400000)	[41, 61)	[130, 170)	[40, 80)	Have allergies	Chinese	M	S79139G
[100000, 400000)	[41, 61)	[130, 170)	[40, 80)	Have allergies	Chinese	M	S76829S
[100000, 400000)	[41, 61)	[130, 170)	[40, 80)	Have allergies	Chinese	M	T360X5A

Figure 35: Records Generated

The result of the query can also be conveniently downloaded into a CSV file.

9. IoT Crowd Sensing Subsystem

Overview

This subsystem will utilise OpenCV IOT devices to detect the crowd count in a specific room and store it in a database table. This subsystem will also allow patients, doctors, medical staff and researchers to view the crowd sensor count through the web application interface. The crowd count will be displayed to the user as shown below in Figure 36. It will display the crowd count percentage in 10 minutes intervals and the popular times of the day where there is a larger crowd.

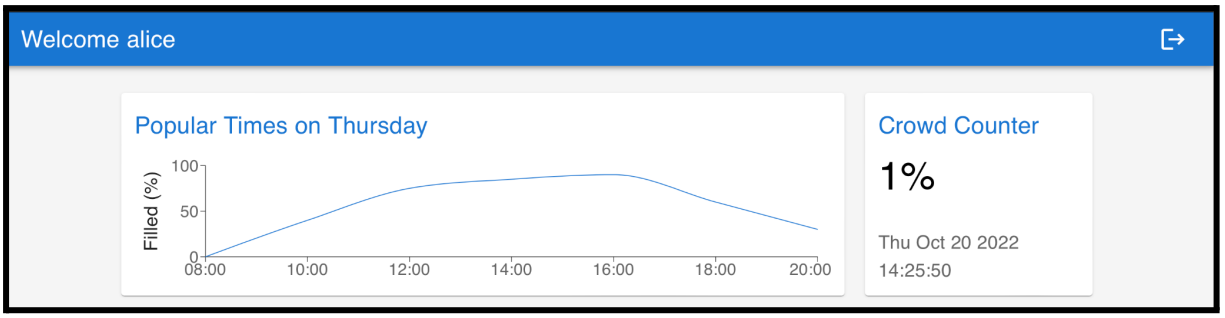


Figure 36: Sample Crowd Counter Shown

9.1 IoT Technologies

Programming Languages & Framework:

- Python 3.9
- Rest API

Additional Libraries:

- Imutils 0.5.4
- OpenCV-python 4.6.0.66
- Numpy 1.23.2

Model Data:

- Common Objects in Context dataset (coco.names)
- Pre trained TensorFlow model on the dataset (ssd_mobilenet_v3_large_coco_2020_01_14.pbtxt)

Devices:

- Raspberry Pi with Camera

9.2 Implementation

OpenCV is a library of programming functions mainly aimed at real-time computer vision. Along with dataset coco.names and a mobile-ssd model intended to perform object detection, a program is set up to perform crowd sensing.

When the program is run on the Raspberry Pi, it will detect the number of people that are present in the room. As the model_data is able to detect different kinds of objects, this program is programmed in a way that it only captures' ‘person’’ objects.

```
if classLabel == "person":
    cv2.rectangle(
        image, (x, y), (x + w, y + h), color=classColor, thickness=1
    )
```

Figure 37: Sample source code for detecting ‘person’ object

This detection program captures frames in 10 minutes intervals and returns the count of the number of people detected in the frame. In this case, Constants.TOTAL_TIME is set to 600 seconds which is equivalent to 10 minutes.

```
total_seconds = Constants.TOTAL_TIME  # timer
while total_seconds > 0:
    time.sleep(1)
    total_seconds -= 1
    key = cv2.waitKey(1) & 0xFF
    if key == ord("q"):
        isLoop = False
        break
```

Figure 38: Sample source code for setting time interval

9.2.1 Interaction with Database

Using the REST API, a post request is sent to the backend server where it will process the data and store it in the database. This occurs in 10 minute intervals. This record will store the timestamp in the (YYYY-MM-DD HH:MM:SS +8) format followed by the crowd count as an integer. The primary key for the table will be the timestamp. The json data that will include ("time_recorded": formatted_time) and ("count": count) where formatted_time is the timestamp as mentioned above.

Afterwhich patients, doctors, medical staff and researchers will be able to view the crowd count. The backend obtains the crowd count from the crowd data table. Then, the frontend server obtains the crowd count from the backend server and displays it to the user as shown previously in Figure 36.

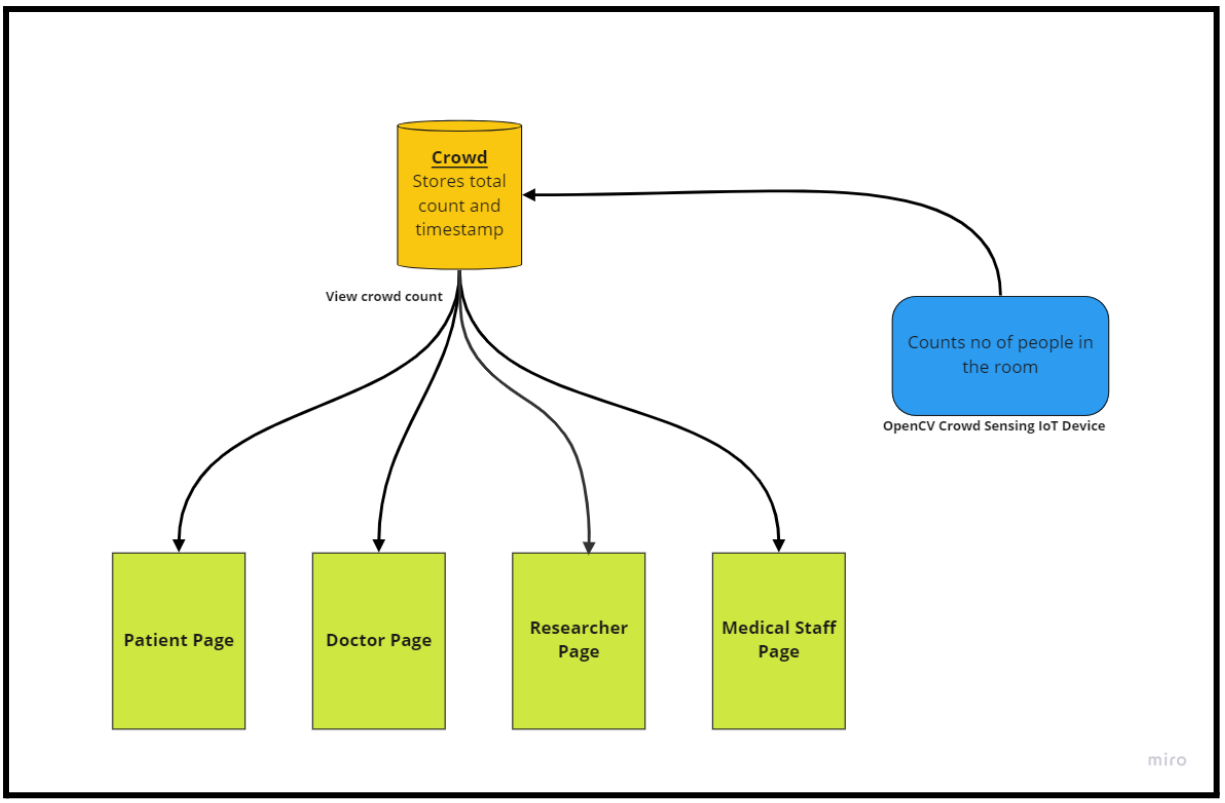


Figure 39: Crowd sensor Subsystem design

9.3 Security implementation

The connection between the IOT device and the backend is secured by SSL. Hence, the data that is transmitted from the IOT device to the backend is encrypted and more secure. Communication is safe from attackers who are trying to eavesdrop or enable other man-in-the-middle attacks.

In addition, an API_token is stored in the IOT device. The backend will verify if the token is valid before accepting the POST request. This prevents any other malicious users from sending POST requests and altering the actual data. To ensure forward secrecy, a one-way hash function with a random generated seed is used. SHA-256 is used to hash the token concatenated with the randomly generated seed before sending it to the backend. The randomly generated seed is generated using os.urandom() where it is a cryptographically secure generated seed. The client-side will send the computed hash and seed to the backend. Thereafter, the backend will obtain the seed, compute the hash with the server API_token and compare with the received hash from the client side. If the hash is the same, the post request is accepted. This prevents an attacker from obtaining the API_token while performing eavesdropping or man-in-the-middle attacks.

10. Database

Overview

The database is used to store all user account information, patient's health records, examination results of a patient visit, and anonymized health records. Certain Django plugins that are used also require its own table to function properly. Such tables have been omitted in figure 40 below for simplicity.

10.1 Implementation

The database is hosted on an Ubuntu virtual machine running PostgreSQL 12.12. The connection between the backend and the PostgreSQL server is handled automatically by Django when the backend requires database access. Django is also configured to use SSL for the connection to prevent information leakage through eavesdropping.

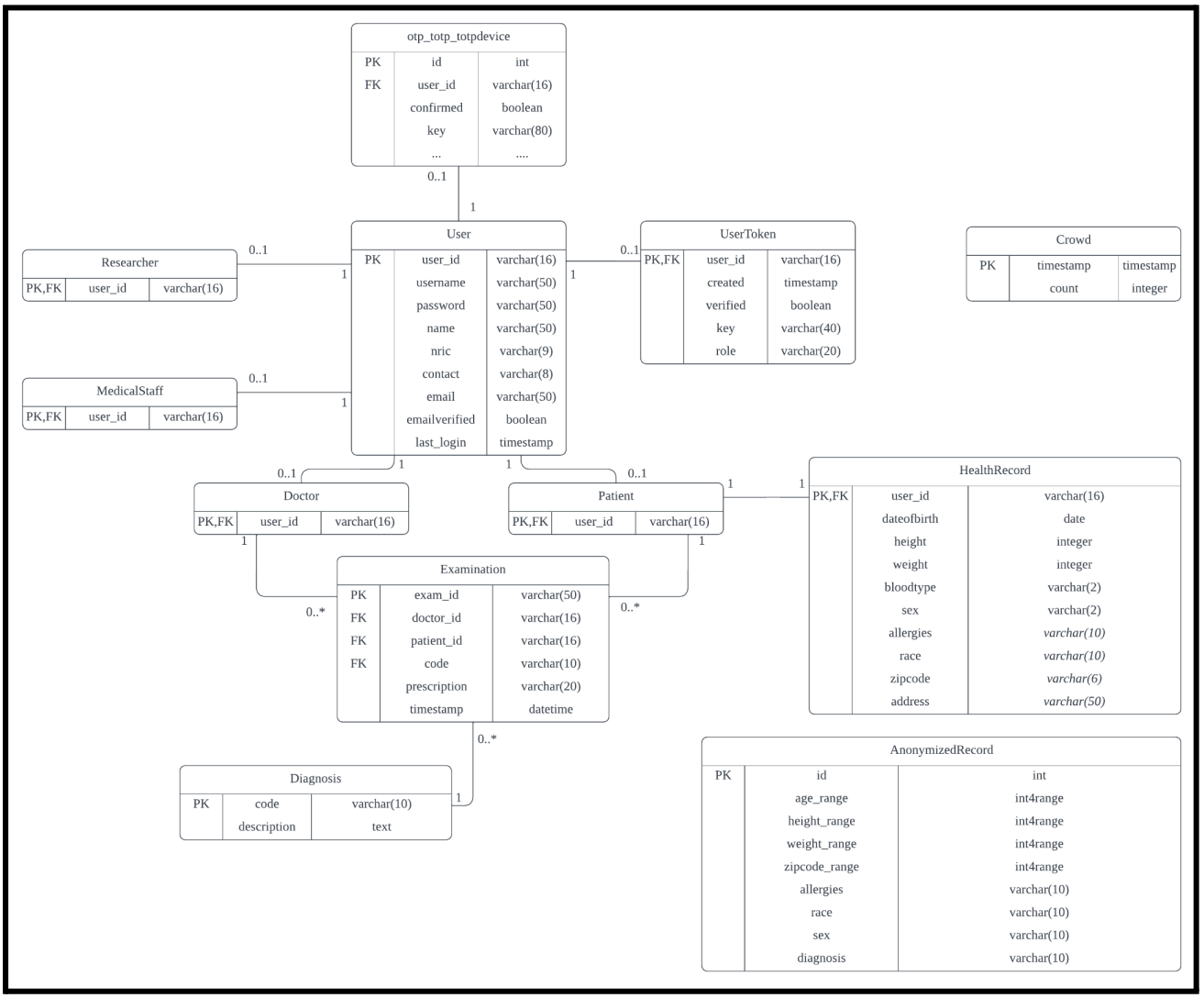


Figure 40: Database ER Diagram

The section below describes some important tables and their fields.

User Table

Field	Purpose
user_id	Randomly generated unique string used to identify a user
username	Used by a user to login
password	Password of the user derived with PBKDF2
email	Email address used when a user requests to remove an OTP device
last_login	Timestamp of a user's last login
email_verified	By default, email_verified is set to be true as we assume the patient will give a valid email when registering for an account at the facility

Patient / Doctor / MedicalStaff / Researcher Table

Field	Purpose
user_id	If a user_id exists in a table, the user is assigned to the role corresponding to the table name. A user can have more than one role.

UserToken Table

Field	Purpose
user_id	Identify which user the token belongs to
time_created	Timestamp of the token creation
key	Randomly generated 256-bit hex-string
verified	Whether the user has verified the token with an OTP.
role	The role that the user logged in as

otp_totp_totpdevice Table

Field	Purpose
user_id	Identify which user the device belongs to
confirmed	Whether the device has been used to verify an OTP at least once.
key	Secret key generated that is shown to the user during creation. The user has to pass this key to the mobile authenticator app via QR code or manual input.
…	The **django-otp** library automatically creates some extra fields for OTP

	generation which are omitted here

Diagnosis Table

Field	Purpose
code	ICD-10 code used by doctors during patient examinations
description	Description of the diagnosis

Crowd Table

Field	Purpose
timestamp	Timestamp of the IOT count
count	The number of people the IOT device captured and recognised

10.2 Generating data

To generate data, a script was created using the python **faker** library. The library is used to generate fake names and addresses.

By default, accounts are created manually and the password stored in the database is derived using PBKDF2. However, when creating thousands of accounts using the script, the time and computational power required forced us to look for an alternative method. Since accounts generated by the script need not be accessible, the script simply generates a random hex string and appropriately formats it.

The patient examinations are generated by randomly selecting a patient, doctor and a diagnosis code. This is fairly inaccurate as the distribution of diagnosis is not uniform. However, the aim of the generated data is to produce enough samples for a k-anonymous database so this was accepted by us.

All data generated are written to a CSV file which is easily imported into PostgreSQL using the COPY command.

Here is a short summary of how certain data is generated.

User Table

Field	Source
user_id	Generated hex string using **python.secrets**
password	Generated base64 encoded string using **os.urandom**

name, email	Generated with **faker** library

Health records

Field	Source
allergies	Eggs | Milk and Dairy | Peanuts | Tree nuts | Fish | Shellfish | Wheat | Soy | Sesame | None
race	Chinese | Malay | Indian | Others
dateofbirth	1930-01-01 — 2015-02-01
weight	40 - 110
height	130 - 200
zipcode	100000 - 900000
address	Generated with **faker** library

Examinations

Field	Source
diagnosis_code	Selected from 70000 ICD-10 diagnosis code
exam_time	2018-01-01 — 2022-02-01
patient_id	Selected from pool of patients
doctor_id	Selected from pool of doctors

10.3 Security Implementation

The connection between the backend and PostgreSQL is secured with SSL. We use a self-signed certificate for SSL since the database VM is within the SoC network and is inaccessible by users outside the SoC network. As the database communicates with the backend only, we set the backend server to accept and use the self-signed certificate for SSL connections. Communication is thus encrypted and safe from eavesdropping from malicious actors residing within the network.

The virtual machine hosting the PostgreSQL server is also hardened by using **iptables** to restrict incoming traffic with the exception of port 22 for ssh and port 5432 for PostgreSQL.

11. DevOps

Overview

For CI/CD tools, we utilised GitHub actions as a platform to integrate CI/CD into our project. It is a platform to allow automation of build, tests, and deployment pipeline. Workflows are created to build and test automatically on push and pull requests to the main branch. Pre-commit actions are also integrated to ensure that newly committed files are automatically formatted according to programming standards before being committed to GitHub.

11.1 Pre-commit

Black Formatter

Black is a PEP8 compliant opinionated formatter for python code.

Prettier Formatter

Prettier code formatter is an opinionated code formatter that supports many different languages. In this case, it formats JavaScript, HTML, CSS, YAML files.

11.2 Continuous Integration (CI)

Nodejs

The correct dependencies and libraries are installed on the runner. The Nodejs project is built and tests are run. Upon receiving any errors with the build or tests, warnings will be thrown and changes have to be made before merging and deploying.

Django CI

First, the correct dependencies and libraries are installed on the runner. The Django project is then built and tests are run on the project. Any errors building or testing the code will be caught and need to be rectified before being merged and deployed on the VMs.

Python

The correct dependencies and libraries are installed, tests are run with python files and lint is run with the version of python we are using.

Black formatter

The correct version of black is installed and black formatting check is run across all the files in the repository.

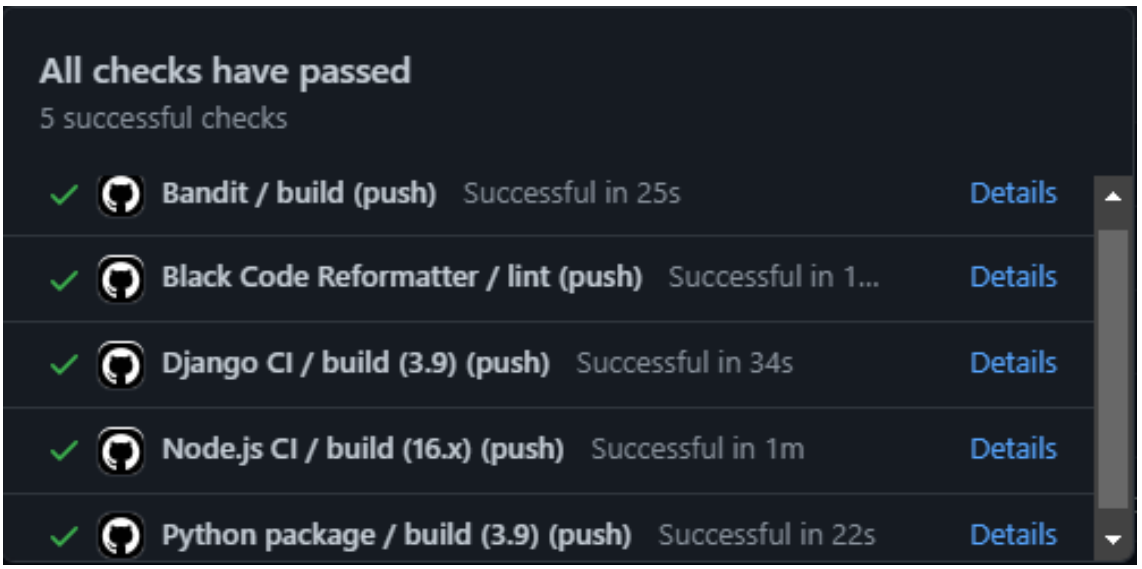


Figure 41: Example of the Github Action checks during a push to the main branch

12. DevSecOps

12.1 Software Composition Analysis (SCA)

Dependabot

Dependabot checks dependency files in our repository to ensure that our dependencies are up to date and free from known vulnerabilities. It regularly checks dependency files for outdated requirements and alerts us whenever an outdated dependency is vulnerable to known attacks.

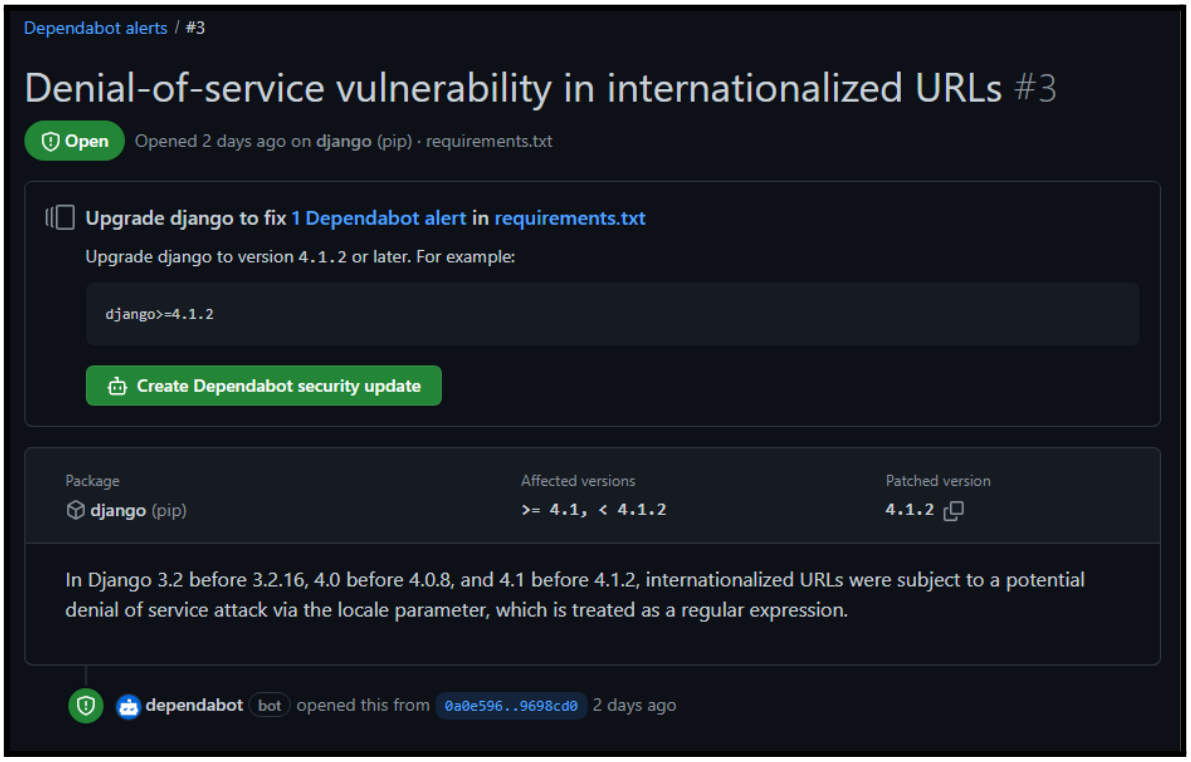


Figure 42: Example of a Dependabot alert

12.2 Security application security testing (SAST)

Bandit

Bandit is a static application security testing (SAST) tool, also known as a security linter, used to analyse Python source code. It is used to find common security issues in Python code in our Django project, as well as the Python code used by our IoT subsystem.

```
Test results:
>> Issue: [B106:hardcoded_password_funcarg] Possible hardcoded password: '12patient12'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   Location: ./backend/test_backend.py:11:20
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b106_hardcoded_password_funcarg.html
10          def setUp(self):
11              self.user = get_user_model().objects.create_user(
12                  username="patient", ***, email="test@example.com"
13              )
```

Figure 43: Example of a Bandit alert

13. Security Claims

This section details the security-related claims that our team is confident of providing to users. We have divided our security claims into four main sections: Server and Infrastructure, Web Security, Functional Security and IoT Security Claims. There are a total of 13 security claims.

13.1 Server and Infrastructure Security Claims

This subsection contains claims where security mechanisms are provided by servers and infrastructure components.

13.1.1 TLS/SSL

Our web application uses TLS implementation. Hence, it is not possible to perform sniffing or man-in-the-middle attacks on the following connections due to the presence of an SSL certificate:

- Connection between client and backend server
- Connection between backend server and database server
- Connection between IoT device and backend server

13.1.2 Access

It is not possible to access any systems or services that are not intended to be accessible. The database server is configured to only accept connections from the backend server via PostgreSQL. The backend server will only accept connections with a valid access token. IoT will only accept connections with the backend server. We use an uncomplicated firewall (ufw) to deny connections to ports that our VMs are not using; only the required ports accept an incoming connection to allow the application to run.

13.2 Web Security Claims

This subsection contains claims where security mechanisms are provided by web application frameworks.

13.2.1 SQL Injection

It is not possible to perform SQL injection on our entire system. Using Django's object-relational mapping (ORM) and query sets to query data allows us to avoid raw SQL queries. Django provides a database abstraction API to query for data in the database using prepared SQL statements. Queries are parameterised and escaped by default, preventing SQL injection attacks.

13.2.2 Cross Site Scripting (XSS)

It is not possible to perform XSS on our entire system. Django escapes specific HTML characters which may produce unauthorised JavaScript execution code, preventing stored XSS. Applying defensive programming where possible allows us to avoid displaying any unnecessary input back to the user to prevent reflective XSS. React also sanitises input by

default as React DOM escapes values embedded in JSX and converts them into strings before rendering, preventing code injection such as in form fields. We also do not render HTML elements directly so there is no use of the React prop `dangerouslySetInnerHTML`, nor do we insert HTML into the DOM directly such as through innerHTML.

13.2.3 Clickjacking

It is not possible to perform clickjacking attacks on our entire system. To prevent clickjacking attacks, which work by tricking users into performing an action in Medibook by disguising Medibook in a malicious site by wrapping Medibook in an iframe, we use the X-Frame-Options middleware to prohibit our pages from being rendered inside an iframe.

13.2.4 Cross Site Request Forgery (CSRF)

It is not possible to perform CSRF attacks on our entire system. With the usage of authorisation tokens, the attacker is unable to set the authorisation header with the valid token of the user. CORS headers are also used on the backend server to explicitly allow only certain domains to connect to it.

13.2.5 Password Management

It is not possible to recover a user's plaintext passwords from our system. We use the Password-Based Key Derivation Function 2 (PBKDF2) algorithm with a SHA256 hash provided by Django as our password hashing algorithm. PBKDF2 is a password-stretching mechanism recommended by the National Institute of Standards and Technology (NIST).

13.3 Functional Security Claims

This subsection contains claims where security mechanisms are provided by our team's source code that has security implications on the security posture of the web application.

13.3.1 Access Control

It is not possible for doctors, patients, staff, or researchers to perform any action outside of their given roles and permissions as defined by the final report. This includes viewing unauthorised information.

13.3.2 Access Token

It is not possible to access backend resources without a valid token. The token is also not easily brute-forced as it is a 256-bit randomly generated string. As traffic to the backend server is encrypted with HTTPS, the token cannot be stolen by an eavesdropper as well.

13.3.3 Identification of Patients

It is not possible for researchers to identify a patient using the anonymised data with a probability greater than 1/10. The anonymised data have been processed to have k-anonymity and l-diversity where k = 10 and l = 5. Thus, when a researcher searches by

any quasi-identifiers, at least 10 rows will be returned and there is at least a 1/10 chance that a researcher has to guess who a patient is.

13.3.4 Modification of Records

It is not possible to modify any records e.g., health records or examination records that have been previously uploaded. There is no such functionality provided in the application and any changes will have to be made directly in the database.

13.4 Internet of Things (IoT) Security Claims

This subsection contains claims where security mechanisms are provided by IoT components.

13.4.1 Man-in-the-middle attack

It is not possible for malicious attackers to perform man-in-the-middle attacks and eavesdrop on the transmission.

13.4.2 API Token

It is not possible for anyone to send IOT POST requests to the backend VM. Only authorised devices with the correct API token can send POST requests to the backend VM. Attackers will not be able to get the API token from the transmission of data.

14. Appendix

Project Link

Website: https://ifs4205-group3-webapp.comp.nus.edu.sg/

GitHub Links

Main Repository:
https://github.com/IFS4205-AY2223-Group3/IFS4205-Group3-Crowdsensing-Med-App

Frontend:
https://github.com/IFS4205-AY2223-Group3/IFS4205-Group3-Crowdsensing-Med-App/tree/main/frontend

Backend:
https://github.com/IFS4205-AY2223-Group3/IFS4205-Group3-Crowdsensing-Med-App/tree/main/backend

IoT:
https://github.com/IFS4205-AY2223-Group3/IFS4205-Group3-Crowdsensing-Med-App/tree/main/IOT

Database:
https://github.com/IFS4205-AY2223-Group3/IFS4205-Group3-Crowdsensing-Med-App/tree/main/database

Documentation Links

Tool Assessment Report:
https://docs.google.com/document/d/e/2PACX-1vTGhCyDKA7bnu0evuFmf81IEtbelG8YUrHI5V-TGfqEQGkW1FVproLme3CIbL624vHRnzNMxJKmC_9w/pub

System Design Report:
https://docs.google.com/document/d/e/2PACX-1vTkA7unEwErcJg6CfcjvEJu6oonRQkccKQd96N0LHG9S4bAt8dbymTHUYctXwW6y-RI5jHTE3VuWwDV/pub

Database Design Report:
https://docs.google.com/document/d/e/2PACX-1vRr58OstDa70E4Fcrb6f-FAEFH1gXMiOqc-AO0m59QY0cZL_98RLhtUMyj9wXOzxbzmRsoMiVnSt1If/pub

Additional Links

API Document:
https://docs.google.com/document/d/e/2PACX-1vSM7y25c5IF41MuNPW7LGo7hoZ6DTaCSP8X78IYRP81L-qzVRfifYq-S1fFUDILC5gjEG7FZjabQdo-/pub

Miro Board:

https://miro.com/app/board/uXjVPdgPY9o=/?share_link_id=266596687450

Project Timeline Gantt Chart:
https://docs.google.com/spreadsheets/d/e/2PACX-1vR2sbloFGrw1HQKt5EJd9P41jbXM-oE8rRfcLQqsAWr8bbvnx2JsH1_trn8qcFWEw/pubhtml