LAB University of Applied Sciences

My Medi: Web-Based Admin Dashboard for Medication Management

An Admin Tool to Help Monitor User Activity and Maintain Digital Health Services

LAB University of Applied Sciences
Bachelor of Engineering, Industrial Information Technology
Ahanaf Adil

Abstract

Author(s)	Publication type	Completion year
Ahanaf Adil	Thesis, UAS	2025
	Number of pages	
	36	

Title of the thesis

My Medi: Web-Based Admin Dashboard for Medication Management

An Admin Tool to Help Monitor User Activity and Maintain Digital Health Services

Degree, Field of Study

Bachelor of Engineering, Industrial Information Technology

Name, title and organisation of the client

Abstract

This thesis describes the development of My Medi, a web-based administration dashboard designed to enhance medication management for elderly patients within digital healthcare systems. The platform leverages Flask for backend operations and Firebase for secure authentication and real-time database management. And hospital administrators to oversee users, monitor activities, and generate QR codes for prescription verification all while maintaining patient privacy by limiting access to sensitive medical data. Key features include user and role management, activity logging, system health monitoring, and the enforcement of access controls. Through a series of structured manual tests, the system demonstrated reliability, usability, and compliance with core healthcare IT requirements. Comparing the features of the system to contemporary solutions such as Medisafe and MyTherapy revealed that My Medi offers enhanced administrative transparency, supports fine-grained role separation, and integrates patient verification using QR codes features that are limited in the existing platforms. The results established that My Medi have a practical and scalable approach. which bridge the gap between accessibility and secure administrative control in digital medication management. However, its effectiveness in real-world scenarios would require further validation in healthcare settings.

Keywords

Medication adherence, Admin Dashboard, QR code, elderly care, digital health systems, web app

Contents

1 Introduction

The aim of this thesis was to develop a web-based administrative system called My Medi, designed to support the coordination and management of medication adherence among elderly patients. The application assists administrators in overseeing doctor and patient accounts while indirectly supporting patients in following their prescribed medication schedules. Medication adherence has been widely recognized as a key factor in achieving positive health outcomes, especially among elderly individuals who often manage multiple medications daily. Non-adherence has been linked to increased hospitalizations, elevated healthcare costs, and a rise in preventable complications (Banning 2008, 1550–1561, Osterberg and Blaschke 2005, 487–497)

Existing studies have shown that many older adults still rely on handwritten notes or assistance from family members to remember their medication routines, despite the inherent unreliability of such approaches (Banning 2008, 1550–1561). This issue has driven the search for more structured and digital solutions capable of reducing user error and supporting independent adherence monitoring. Digital health technologies have been found effective in improving patient adherence, providing real-time tracking, and enabling healthcare providers to intervene proactively when doses are missed (Thakkar et al 2016, 340–349). The proposed My Medi platform seeks to integrate these technological benefits into a practical system for hospitals and care units. The system has been developed with the understanding that real-world healthcare environments demand tools that are not only technically sound but also usable by non-technical staff and adaptable to patient needs.

The motivation behind this project was partly shaped by personal observations and professional insights into the challenges of managing long-term medication regimens among older adults. Research confirms that involving stakeholders and understanding contextual barriers can lead to more sustainable digital health interventions (Greenhalgh et al 2017, e367).

The challenge is to observe the medication adherence among elderly patients, with studies indicating that over 40% of older adults commit medication-related errors, leading to increased emergency visits and health complications (Gellad et al 2011, 11–23). This problem is exacerbated when patients are left to manage multiple medications alone, particularly in home settings without consistent supervision (Osterberg and Blaschke 2005, 487–497). Although digital tools like Medisafe and MyTherapy offer reminder systems and health tracking features, limitations persist. Medisafe has been designed to send timely reminders, yet it lacks any form of interaction or monitoring capability for doctors (Medisafe 2023). Similarly, MyTherapy offers extensive tracking options, but its user experience has often been

found overwhelming for elderly users, especially those less familiar with smartphones (MyTherapyApp 2021). These drawbacks underline a gap in the current digital health landscape—specifically the need for a unified, accessible, and interactive solution tailored to the needs of older adults and healthcare professionals.

In response to these issues, this thesis aims to develop a web-based administrative system integrated into the broader My Medi platform. The objective is to offer a secure and user-friendly user experience that enables hospital or clinic administrators to manage doctor and patient accounts, oversee prescription management workflows, and maintain system reliability without accessing confidential medical records. Additionally, this system is intended to support core operations such as QR code generation, activity monitoring, and role-based access control through the use of Firebase and Flask technologies. Through this approach, it is expected that the platform will contribute to improving medication adherence and enhancing operational efficiency in healthcare environments.

This thesis report includes six chapters. The second chapter (Operational Environment) provides an overview of the current healthcare system and examines other applications that have been developed and what would be required to implement the solution. Chapter three (Tools and programs) discusses the tools and technologies applied (Flask and Firebase) for the application. The fourth chapter (App Designed and Built) explains the design process and construction of the application, including the structure, user experience, core functionalities, and how various features were developed and connected to form the final working system. The fifth chapter (Testing and Results) presents how the application was rigorously tested, including specific usability, functionality, security, and results, demonstrating the system's reliability and effectiveness. The last chapter (Conclusion and Future Development) overviews the project results and some suggestions for further development.

2 Operational environment

2.1 General overview

The My Medi project is developed to improve the medication process among elderly patients. It has been well-documented that non-adherence to prescribed medication regimens leads to increased healthcare expenditures, preventable hospital admissions, and worsened health outcomes (Cutler 2018). To provide a digital solution, My Medi was designed with a centralized Firebase backend serving as the core of the system. Both the mobile application and the web-based administration dashboard connect to this backend, which manages data storage, user authentication, and real-time synchronization across all platform components. Figure 1 illustrates the high-level system architecture, outlining how each component interacts with the centralized backend to provide a cohesive medication management platform.

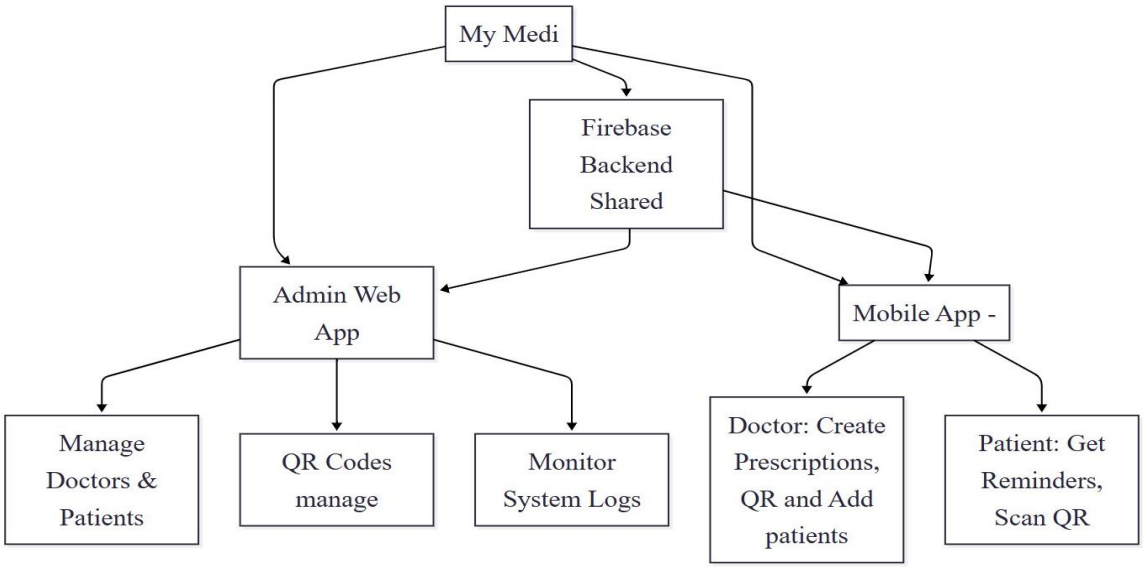


Figure 1. General working process of My Medi

A unified mobile application will be implemented, featuring a role-based login system to accommodate both doctors and patients within a single platform. Upon authentication, the dashboard and available functionalities are dynamically adjusted based on the user's assigned role. Doctors are granted access to create and manage prescriptions, including dosage, timing, and duration. Once a prescription is submitted, a unique QR code is automatically generated and linked to that prescription. These QR codes are intended to be printed and affixed to medication packaging, allowing patients to verify and confirm medication intake through scanning.

Patients, upon logging into the application, are provided with medication reminders scheduled according to the prescription details. A QR code scanning feature has been incorporated to enable confirmation of medication intake. The webapp design aimed to provide a simple and user-friendly, particularly to meet the needs of elderly users with limited technical skills or cognitive challenges. This approach ensures a consistent user experience while minimizing development overhead and promoting accessibility (Barros 2014). Both mobile applications were developed by a project partner and are integrated with the same Firebase backend to maintain synchronized data access.

The web-based administration dashboard, which constituted the core focus of this thesis project, was designed as the control layer of the My Medi ecosystem. This dashboard is used by administrators to manage both doctor and patient user accounts, provide technical support, and oversee the general health and performance of the system. System operations, such as login attempts, account modifications, and optional QR code generation, can be monitored in real time. Importantly, administrators are granted operational access without visibility into confidential medical records, ensuring strict data segregation and compliance with privacy regulations.

2.2 Existing technology

Several digital healthcare solutions are currently available to assist with medication adherence, each offering unique advantages and facing specific limitations. One of the more popular applications is Medisafe, which sends timely medication reminders and allows users to manage their dosage schedules effectively. However, it lacks direct interaction between patients and healthcare providers. Doctors are unable to monitor adherence in real time or adjust prescriptions through the system, which limits its effectiveness in ongoing treatment management (Medisafe 2023).

MyTherapy is another application that enables users to track medication intake, symptoms, and other health data. Despite its wide range of features, the app’s complex dashboard often poses difficulties for elderly users, reducing its accessibility and practical usability for a key target group (MyTherapyApp 2021).

Dosecast offers a simpler experience by focusing on medication scheduling and reminder notifications. While it is user-friendly, it falls short in features like real-time tracking and does not support any verification mechanism, such as QR code scanning, to confirm whether a patient has taken their medication (Dosecast Pill Reminder 2023).

In Figure 2 different types of existing medication application platforms have been introduced. Although these tools serve essential roles in promoting medication adherence, notable gaps remain particularly in healthcare provider integration, ease of use for the elderly, and adherence verification methods. To address these limitations, the My Medi system was conceptualized and developed as a unified mobile and web-based platform. It incorporates user management, backend monitoring, and optional QR-based medication confirmation, offering a robust and accessible solution for patients and healthcare professionals.

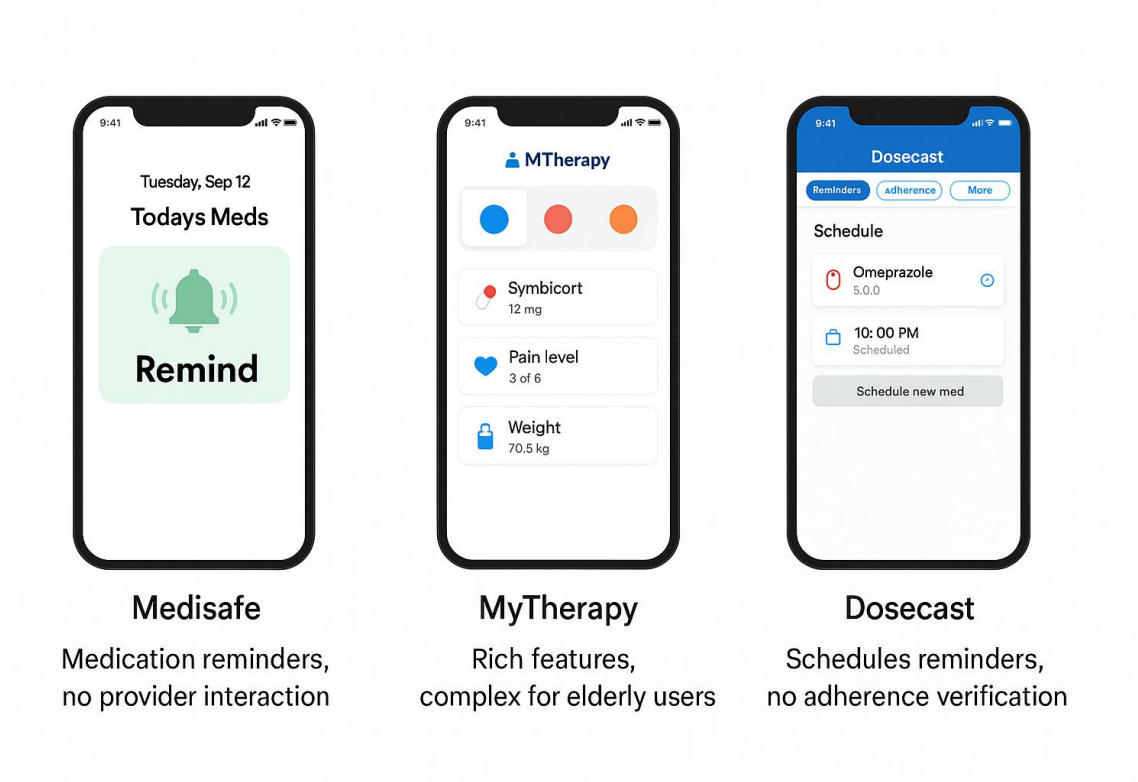


Figure 2. Comparative overview of existing medication reminder applications

In the context of Finland, the national Kanta service has been established as a central repository for patient health information, including prescribed medications. However, Kanta does not actively assist with real-time adherence verification or interactive reminder functionalities, which positions supplementary solutions like My Medi as complementary innovations to the existing infrastructure (Jormanainen 2023, 55–70).

3 Tools and programs

3.1 Flask Framework Overview

In this section, the Flask web framework is described from a theoretical and architectural perspective to establish a solid understanding of the technology used in this project. First, the overall architecture and core characteristics of Flask are introduced, highlighting its role as a lightweight, flexible, and modular Python web development framework suitable for a wide range of web applications, particularly administrative interfaces. Subsequently, detailed explanations of Flask's internal components and essential mechanisms are provided, covering aspects such as routing and view handling, recommended file and folder structures, middleware functionality, and comprehensive security practices, including session management. These foundational concepts help clarify the decisions made in the practical implementation of the admin web application presented in this thesis. The information described in this section provides the reader with sufficient background knowledge required to understand Flask's functionalities, thus clearly separating theoretical foundations from the practical application detailed in subsequent chapters.

3.1.1 Introduction to Flask Architecture

Flask is classified as a micro-framework, primarily built in Python, providing minimal core functionalities required to build a web application. Originally introduced by Armin Ronacher in 2010, Flask has become widely popular for its simplicity, flexibility, and extensibility. Flask's simplicity and flexibility make it ideal for quickly developing and customizing web applications. Unlike heavyweight frameworks, Flask does not include built-in database management, authentication, or advanced form handling by default. Instead, these functionalities can be added through various third-party extensions (Flask Documentation 2024). Flask adheres to the Web Server Gateway Interface (WSGI), which enables seamless interaction between web servers and Python applications. This lightweight and modular architecture promotes rapid development and simplified debugging, facilitating custom solutions tailored to individual project requirements (Flask Documentation 2024, Grinberg 2018, 131–195). Figure 3 highlights key features of web development using Flask, including dynamic app building, lightweight design, Jinja2 templating, and frontend integration

Figure 3. Flask development process (GeeksforGeeks 2025)

3.1.2 Routing and Views

URL routing and views in Flask are implemented via decorators, associating URL patterns to Python functions. When Flask receives an incoming HTTP request, the request URL is matched to corresponding view functions based on defined route decorators. Flask supports both static and dynamic routes, allowing developers to handle dynamic URL parameters directly in the route definition. The request data, including URL parameters, query strings, and form data, is processed by the assigned view functions, generating appropriate HTTP responses in formats such as HTML, JSON, or XML. Furthermore, Flask enables developers to define HTTP methods explicitly, such as GET, POST, PUT, and DELETE, providing clear and precise control over request handling. This routing flexibility simplifies the implementation of RESTful APIs, interactive web interfaces, and complex web applications (Flask Documentation 2024, Norton et al 2005, 483–510).

3.1.3 Flask Folder Structure

The folder structure of the Flask application follows standard modular practices. The project root contains essential files such as the main application script (app.py), configuration files, and core modules including models.py and routes.py. The templates directory holds HTML files for dynamic page rendering with Flask's Jinja2 templating engine. Static assets, including images, CSS, and JavaScript files, are placed in the static directory and its subfolders. Supporting scripts and resources, such as Firebase integration (firebase.py) and QR code files (qrcodes), are also included at the top level. The overall structure is shown in Figure 4.

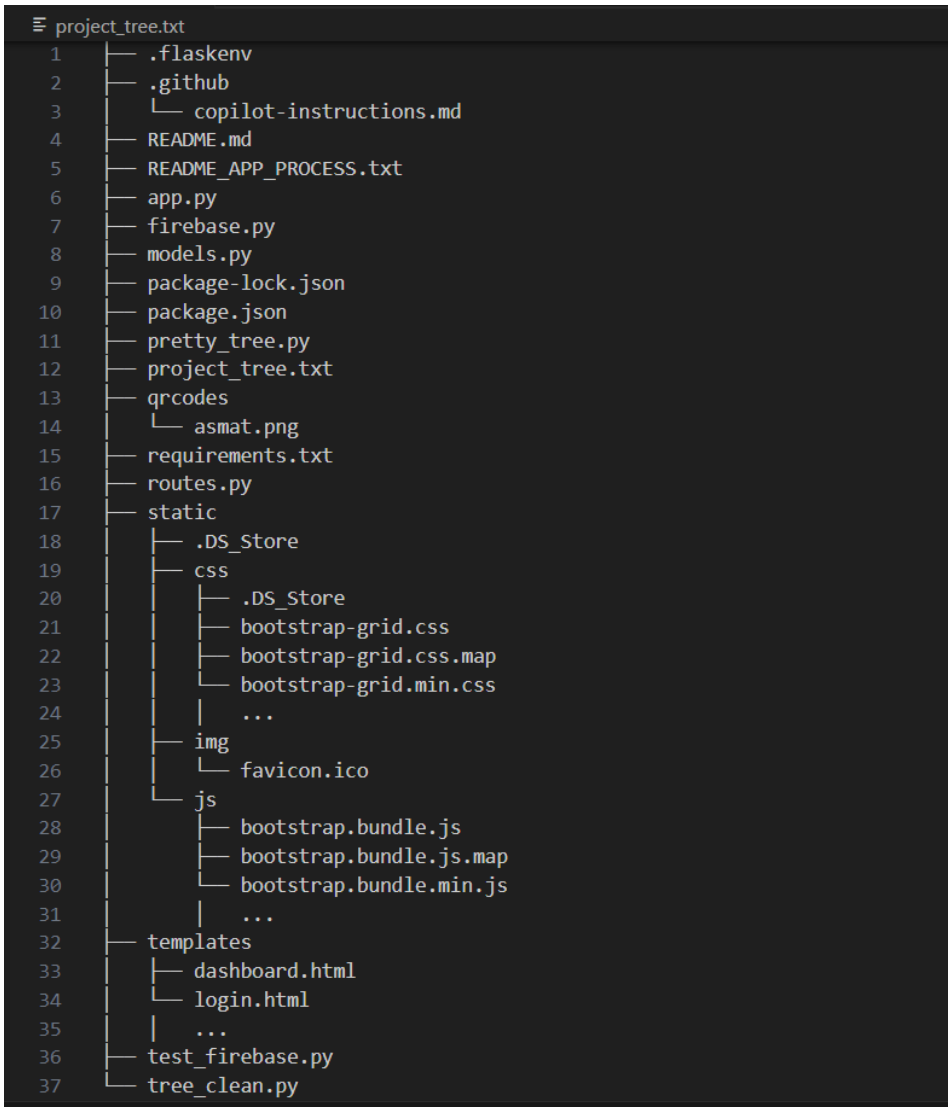


Figure 4. Folder structure of the Flask application

3.1.4 Middleware and Request Lifecycle

Middleware in Flask refers to components or hooks that intercept and manipulate requests and responses within the application's lifecycle. Flask's middleware operates around the WSGI interface, allowing external components to interact seamlessly with the application. Middleware components can handle tasks like request preprocessing, logging, authentication checks, and response post-processing. Flask's request lifecycle consists of several defined stages: upon receiving a request, Flask first evaluates the request context and URL routing. Before view functions execute, middleware such as before_request handlers process incoming requests, allowing for authentication checks or data modification. After the view logic completes execution, Flask executes response middleware (after_request) functions, which can modify outgoing responses, set headers, or perform additional logging. Finally, Flask returns the response to the client and triggers teardown handlers to

manage resource cleanup or error logging. This structured and customizable lifecycle allows developers granular control over application behaviour and response customization (Flask Documentation 2024, Gaspar and Stouffer 2018, 123–180). The flow of request and response processing in Flask applications using middleware is illustrated in Figure 5. The diagram shows how incoming requests pass through a series of middleware components for preprocessing before reaching the main WSGI application, and how responses are similarly processed by middleware during post-processing. This structure allows tasks such as authentication, logging, and header modification to be handled consistently at defined points in the application lifecycle.

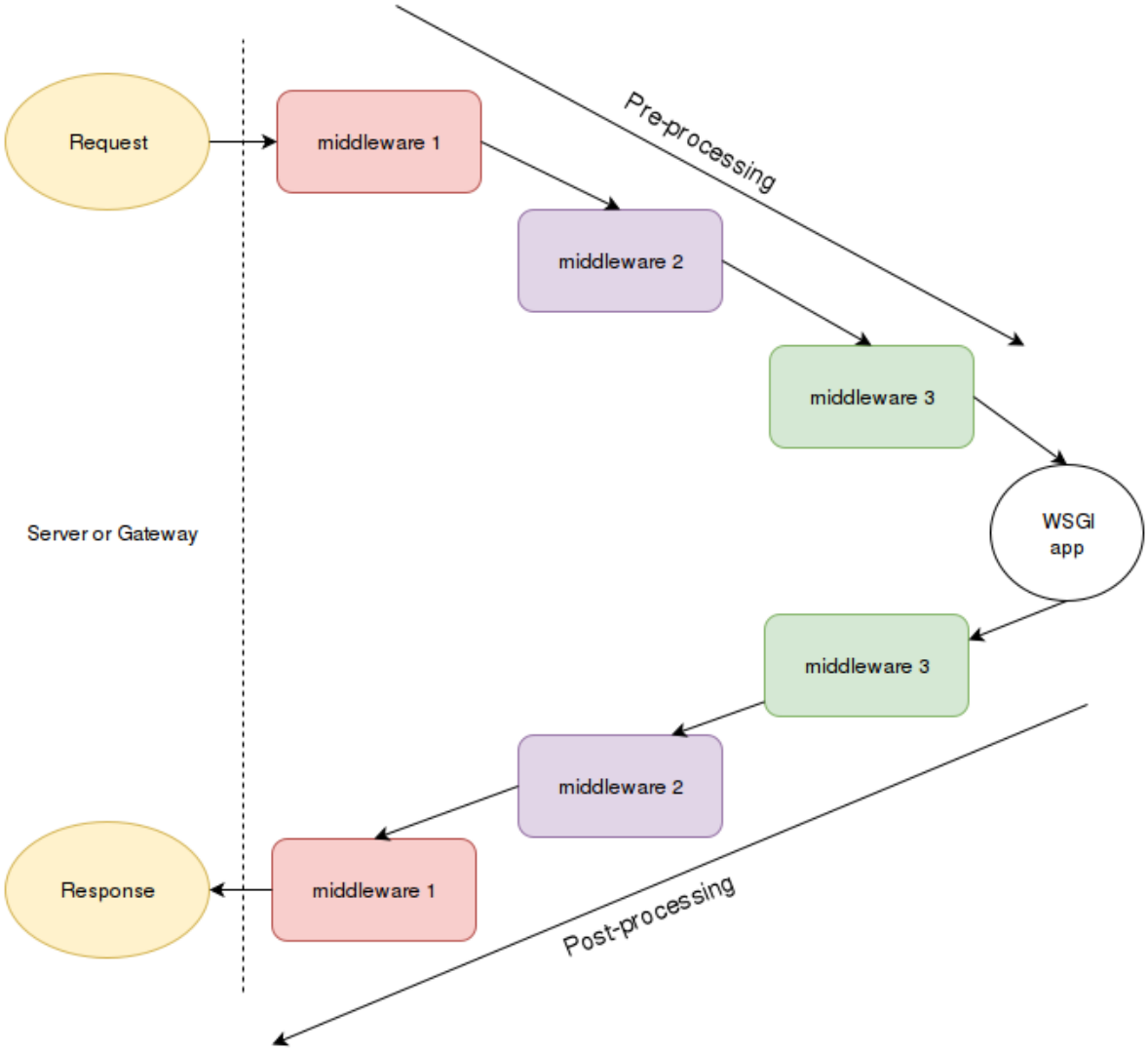


Figure 3. Request and response flow with middleware components in a WSGI-based application (Plone Conf 2016)

3.1.5 Session Management and Security

Session management in Flask involves mechanisms to maintain and track user state across multiple HTTP requests. Flask implements sessions through securely signed cookies stored client-side, ensuring session data integrity by cryptographic signing. For enhanced security, developers commonly integrate extensions such as Flask-Login, which offers user authentication, session handling, and user state management. Flask-Login provides login protection decorators, secure session storage, and seamless integration with database user models.

Security practices in Flask applications encompass various mechanisms to protect against common web vulnerabilities such as Cross-Site Request Forgery (CSRF), Cross-Site Scripting (XSS), and SQL Injection. Flask provides default measures such as secure cookie settings and automatic escaping of potentially harmful user inputs through the Jinja2 templating engine. Additionally, Flask-WTF integrates form handling with robust CSRF protection mechanisms. Extensions such as Flask-Talisman ensure HTTPS enforcement, implementing robust security headers that protect against various browser-based vulnerabilities. Employing these comprehensive security practices ensures applications adhere to modern web security standards, which reduce the risk of vulnerabilities and unauthorized access (Flask-Login 2024, Flask-Security-Too 2024)

3.2 Firebase Overview

In this section, Firebase technology is explored theoretically, providing a solid understanding of its role and functionalities in contemporary web application development. Firebase, developed by Google, is widely recognized as a comprehensive backend platform, offering diverse features critical to modern software development, particularly in web and mobile applications. The following subsections deliver an in-depth overview of Firebase's essential components, including a general introduction to Firebase services, detailed explanations of authentication and authorization mechanisms, the structure and operational features of Firestore database, and an examination of Firebase Software Development Kits (SDKs) and REST APIs. Each subsection elucidates the theoretical foundations, technical characteristics, and benefits that Firebase provides, distinctly separating theoretical concepts from practical applications presented later in the thesis. This foundational knowledge ensures readers fully comprehend Firebase's capabilities and suitability for securely managing user data, authentication processes, real-time database synchronization, and seamless integration within various software ecosystems.

3.2.1 Firebase Features Overview

Firebase is recognized as a comprehensive Backend-as-a-Service (BaaS) platform. It was originally introduced in 2011 as an independent company and was acquired by Google in 2014, after which its product offerings were expanded and more deeply integrated into Google Cloud services. This Figure 6 illustrates Firebase's real-time synchronization capability across multiple devices. Data updates from the cloud are instantly reflected on all connected clients, such as smartphones, tablets, and desktops.

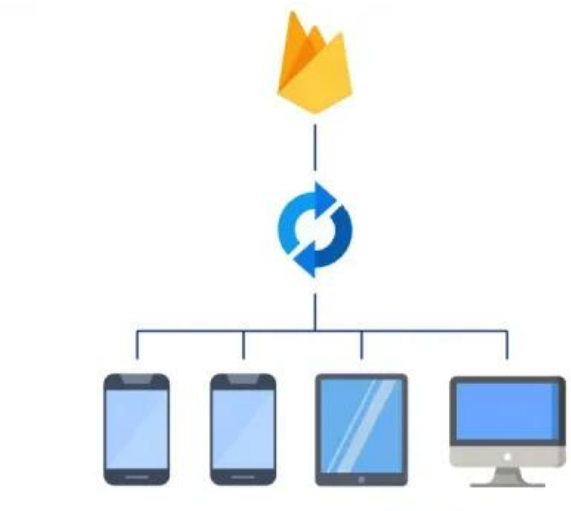


Figure 4. Firebase synchronizes across web and mobile devices (Yahiaoui 2017, 23–55)

The platform provides various backend infrastructure services essential for rapid and scalable application development, especially suitable for web and mobile environments. The major Firebase services include Authentication, Cloud Firestore, Realtime Database, Cloud Functions, Cloud Storage, Hosting, Analytics, and Cloud Messaging, among others. These features collectively enable developers to efficiently manage user authentication, real-time data synchronization, file storage, application hosting, and analytics without extensive server-side code. Firebase’s cloud-based architecture supports automatic scaling and real-time data updates, enabling applications to handle extensive user bases and interactions seamlessly (Google Firebase Documentation 2024, Moroney 2017, 25–92).

3.2.2 Firebase Authentication and Authorization

Firebase Authentication serves as a robust and secure user management system that supports multiple methods of authentication, including email/password combinations, phone authentication, social identity providers such as Google, Facebook, and Twitter, and enterprise identity systems via SAML and OpenID Connect. Upon user login, Firebase issues secure tokens, which authenticate subsequent requests securely and efficiently. Firebase

Authentication integrates directly with other Firebase services, allowing developers to implement role-based access control (RBAC) effectively. Developers manage permissions and user roles using security rules, which define precise authorization parameters to protect user data and enforce strict access controls. This streamlined and integrated authentication solution reduces development complexity, improves application security, and enhances the user experience across web and mobile platforms (Google Firebase Documentation 2024, Firebase 2024)

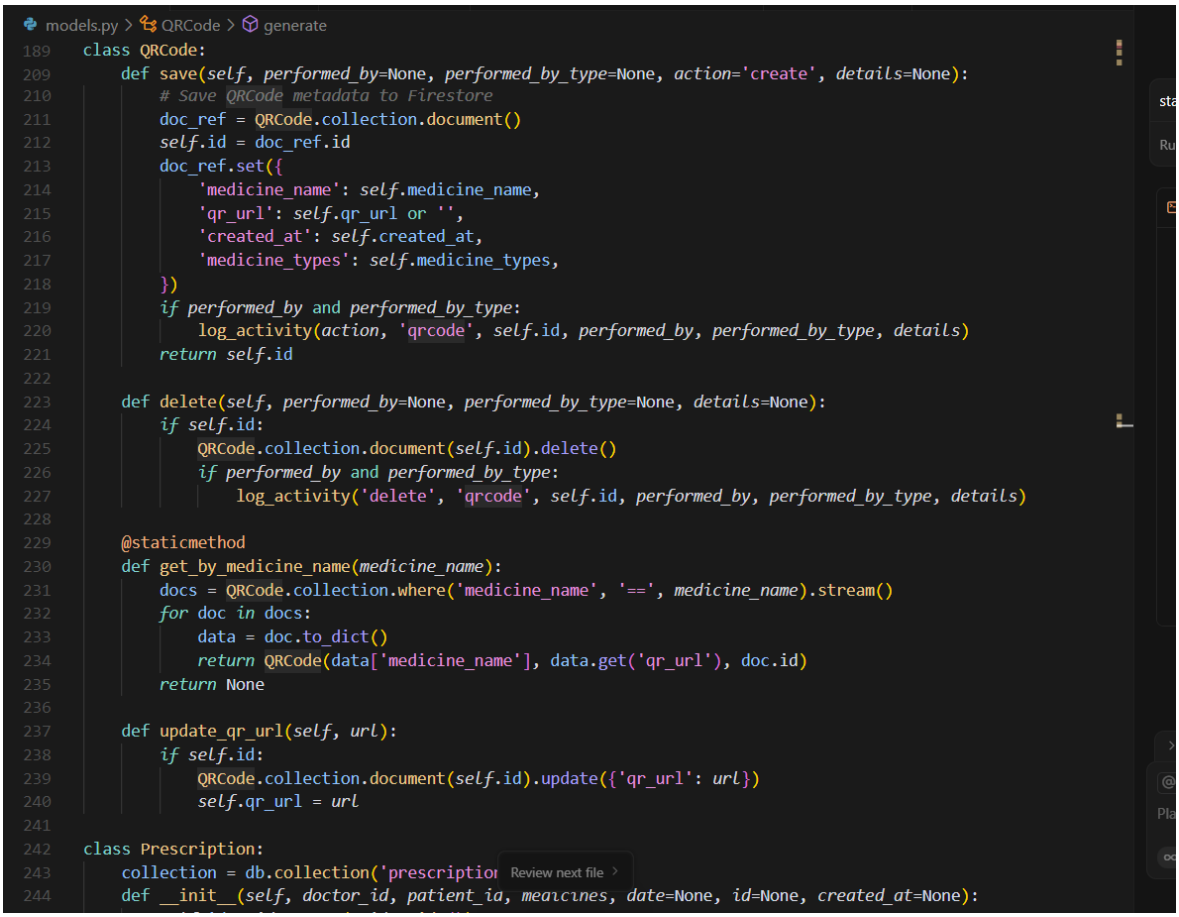
3.2.3 Firestore Database Structure

Cloud Firestore, Firebase's NoSQL database, adopts a flexible and scalable data structure that employs a document-oriented model. Data in Firestore is stored as collections of documents, each consisting of fields mapped to key-value pairs. Documents within Firestore can nest subcollections, providing a structured yet highly flexible approach to data management. Unlike traditional relational databases, Firestore does not require predefined schemas, allowing developers to adapt the data model dynamically based on evolving application requirements. Firestore utilizes real-time synchronization, meaning any changes made to the data are immediately propagated across all connected clients. This functionality enables dynamic and interactive user experiences, particularly beneficial for applications that demand real-time data updates, such as collaborative platforms, chat applications, and monitoring dashboards (Ashok Kumar 2018, 175–231).

3.2.4 Firebase SDKs and REST APIs

Firebase provides Software Development Kits (SDKs) tailored specifically for various platforms and programming languages, including JavaScript, Android (Java/Kotlin), iOS (Objective-C/Swift), Python, and more. These SDKs simplify the integration of Firebase services into applications by providing convenient, platform-specific APIs for Firebase functionalities. Additionally, Firebase offers a comprehensive set of REST APIs, allowing developers to perform Firebase operations using standard HTTP requests, beneficial in server-side contexts and custom integrations where SDKs might not be feasible or desirable. Through the REST APIs, developers can execute operations such as authenticating users, managing data in Firestore, accessing user management capabilities, and sending messages via Cloud Messaging. These versatile SDKs and REST APIs enhance the development flexibility, enabling broad compatibility and streamlined integration of Firebase functionalities into diverse technological environments and application architectures (Google Firebase Documentation 2024).

Firebase provides an official Admin SDK for Python.The following code example demonstrates key CRUD (Create, Read, Update, Delete) operations as implemented in the QRCode model shows on Figure 7. By using the Admin SDK, all data access is securely managed on the server side, minimizing client-side exposure and supporting strict access control.




```python
class QRCode:
    def save(self, performed_by=None, performed_by_type=None, action='create', details=None):
        # Save QRCode metadata to Firestore
        doc_ref = QRCode.collection.document()
        self.id = doc_ref.id
        doc_ref.set({
            'medicine_name': self.medicine_name,
            'qr_url': self.qr_url or '',
            'created_at': self.created_at,
            'medicine_types': self.medicine_types,
        })
        if performed_by and performed_by_type:
            log_activity(action, 'qrcode', self.id, performed_by, performed_by_type, details)
        return self.id

    def delete(self, performed_by=None, performed_by_type=None, details=None):
        if self.id:
            QRCode.collection.document(self.id).delete()
            if performed_by and performed_by_type:
                log_activity('delete', 'qrcode', self.id, performed_by, performed_by_type, details)

    @staticmethod
    def get_by_medicine_name(medicine_name):
        docs = QRCode.collection.where('medicine_name', '==', medicine_name).stream()
        for doc in docs:
            data = doc.to_dict()
            return QRCode(data['medicine_name'], data.get('qr_url'), doc.id)
        return None

    def update_qr_url(self, url):
        if self.id:
            QRCode.collection.document(self.id).update({'qr_url': url})
            self.qr_url = url

class Prescription:
    collection = db.collection('prescriptio
    def __init__(self, doctor_id, patient_ia, meaicines, date=None, id=None, created_at=None):
```

Figure 5. Firebase Admin SDK

4 App design and built

4.1 Designing the app

The My Medi administration web application for the system was developed using simplicity as the main principle, alongside being responsive and secure. This web application is meant for use by hospitals, healthcare institutions, or government health departments to allow authorised personnel to manage doctors, patients, system activity, and account control without being able to see the sensitive prescription data. The app was planned with a clear separation of user roles. While the Doctor and Patient apps are managed through mobile application, the admin dashboard is entirely web-based. It provides central control over system operations but only allows non-medical administrative tasks such as account registration, user management, QR code creation, and system monitoring. Although QR codes are typically generated automatically by the system when a doctor creates a prescription via the mobile app, a fallback feature was included in the admin dashboard to support scenarios where technical issues prevent the doctor from generating the code.

Figure 8 displays the Admin Login Page of the My Medi web-based administration system. This login page is designed to be clean, user-friendly, and focused on security. At the centre of the page, it features a clearly labelled “Admin Login” section that prompts the user to enter their credentials. The form includes two input fields one for the registered email address and another for the password, which also offers a visibility toggle option to ensure correct entry. Below the form is a prominent blue "Login" button, enabling access to the system after successful authentication. Positioned in the top-right corner is a navigation menu button, which likely provides quick access to the dashboard and various administrative sections once logged in. This login page plays a critical role in restricting access to authorized administrators only.

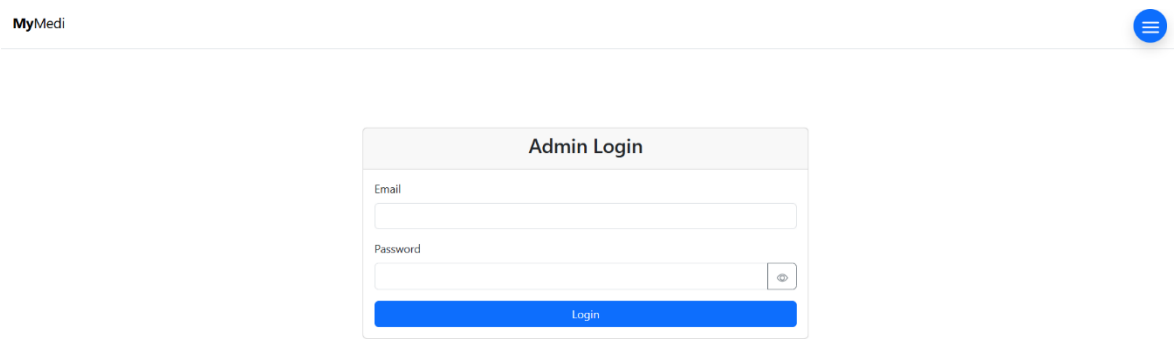


Figure 6. Admin Login page

Figure 9 refers to My Medi Admin Dashboard, which acts as the main platform for healthcare administrators. At the top of the dashboard, the screen displays real-time statistics for key entities in the system, including the number of registered doctors, patients, administrators, QR code prescriptions, recorded activities, and API hits.

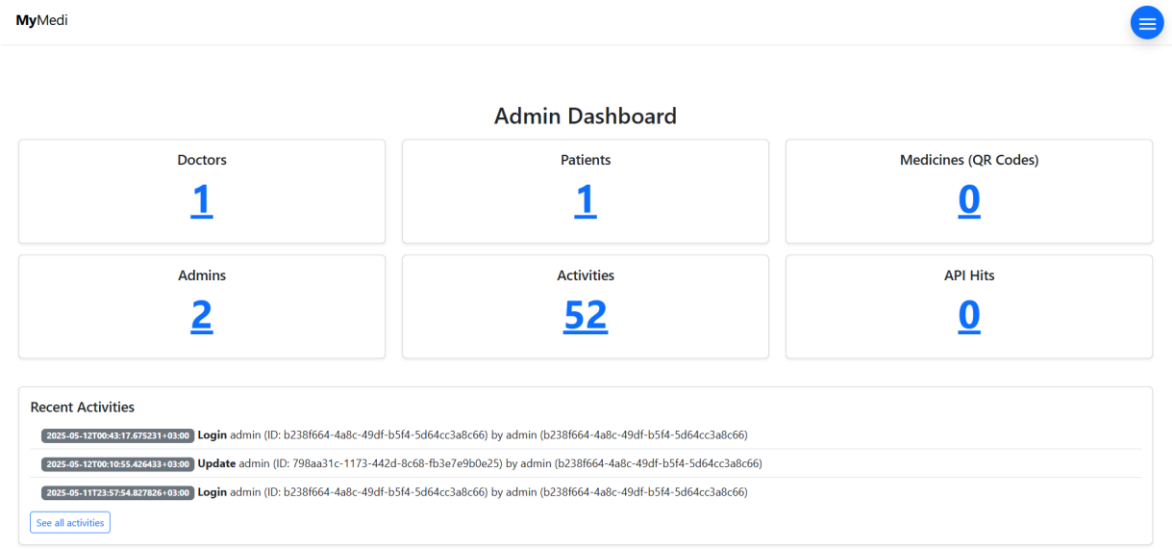


Figure 7. Admin Dashboard page

Each category is represented with a clear label and a dynamically updated value, providing a snapshot of system usage and engagement. Below the statistics, the dashboard includes a Recent Activities section, where the most recent admin actions, such as logins and updates, are listed with timestamps and user IDs. This layout allows administrators to monitor user behavior, maintain transparency, and ensure accountability across the platform. The dashboard's clean, responsive design ensures that hospital IT staff can quickly access essential system metrics and act on them efficiently. Figure 10 shows Doctor User Management line in the My Medi Admin Dashboard. It allows administrators to manage healthcare personnel and patient access efficiently. As seen in the Figure 10, admins can view a list of registered doctors, perform updates, and manage login credentials.

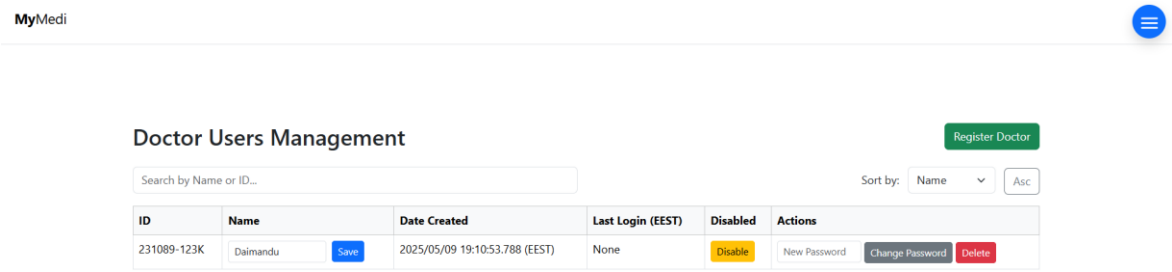


Figure 8. Doctor user management

The dashboard includes an easy-to-navigate panel where each doctor's details are displayed along with action buttons for saving updates, disabling accounts, resetting passwords, or deleting entries. Figure 11 displays the Doctor Registration Form within the system. This form allows administrators to add new doctors by entering their name, email, phone number, unique ID, specialization, experience, and password. These standardized data fields help streamline account handling and reduce technical complexity for hospital staff.

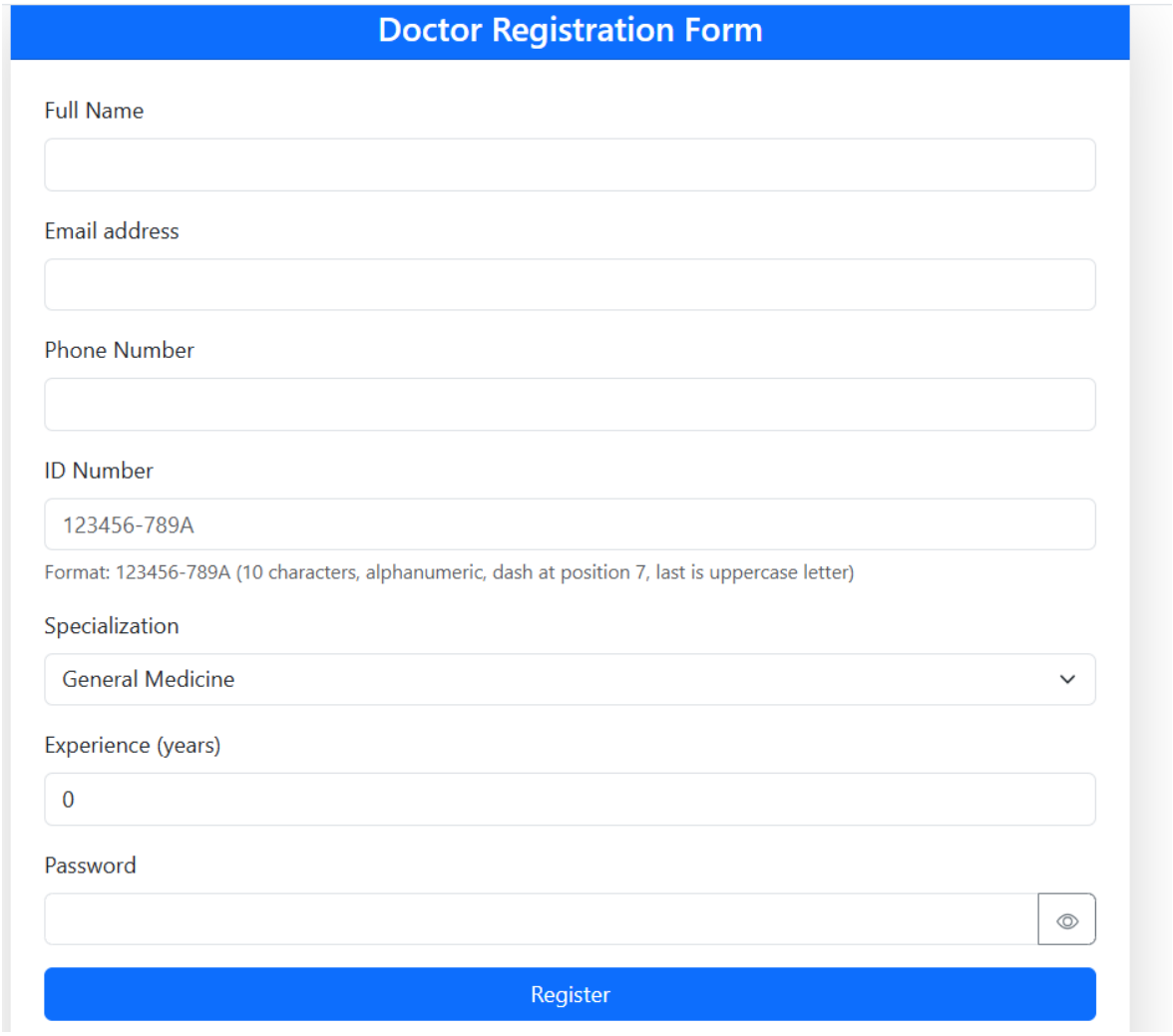


Figure 9. Doctor register form

Figure 12 shows the Patient User Management panel in the My Medi Admin Dashboard, which closely mirrors the functionality of the doctor management platform. This panel enables administrators to manually register patients an especially useful feature for elderly users or those with limited technical skills. This section includes options to search by name or ID, update patient profiles, disable accounts, reset passwords, and delete users as needed. Each patient entry includes action buttons, giving admins full control over account-related

tasks without accessing any medical or prescription data, thus ensuring user privacy and maintaining data protection standards.

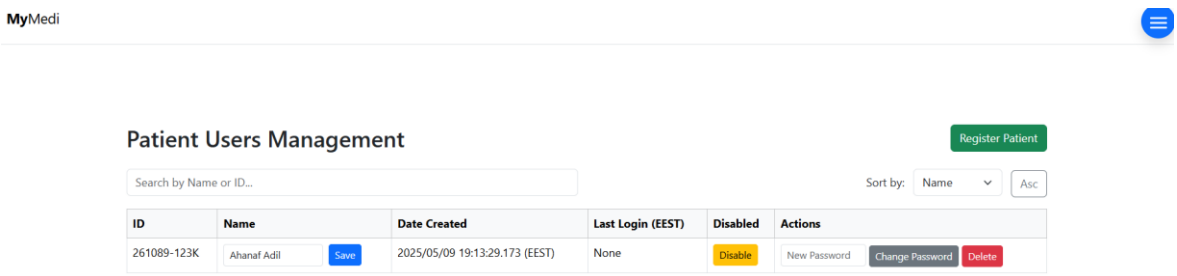


Figure 10. Patient user management

The Patient Registration Form, as shown in Figure 13, is designed to be straightforward and user-friendly. It requires only basic details such as full name, email address, phone number, age, sex, ID number, and password. This streamlined approach ensures that even non-technical staff can complete the registration process quickly and without confusion, reducing the likelihood of errors and improving the overall efficiency of patient onboarding.

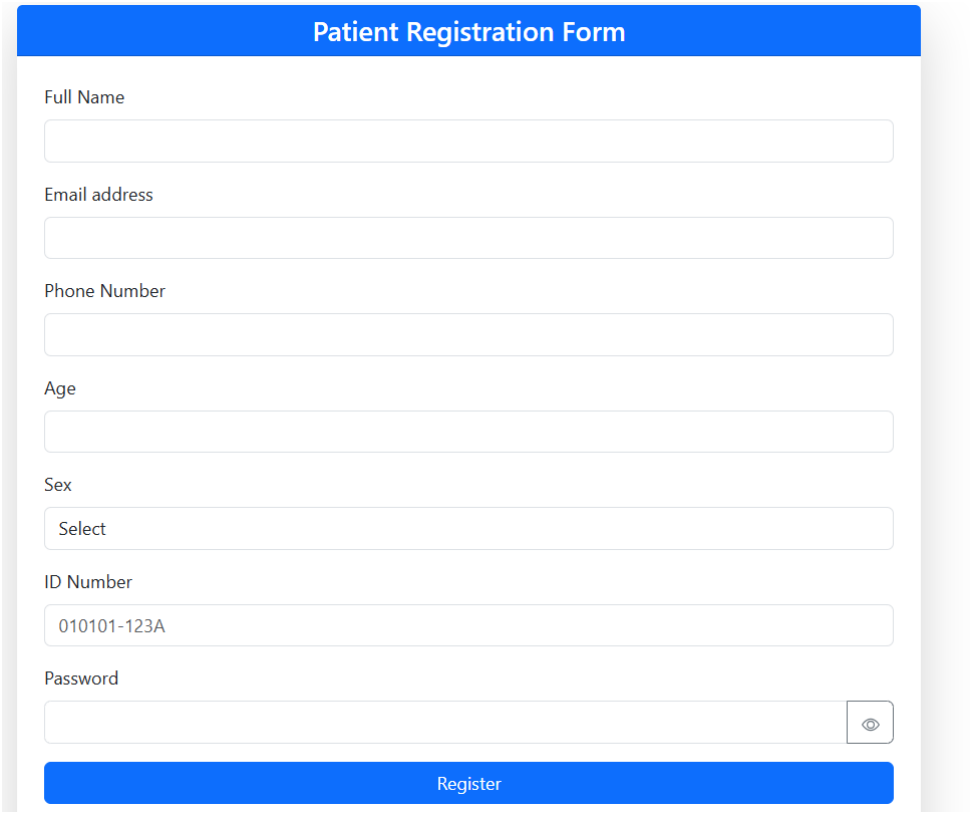


Figure 11. Patients register

Figure 14 shows the QR Code Generator tool within the dashboard, which is designed to be straightforward and efficient. Administrators simply input the medicine name and select

the corresponding medicine type from a dropdown list for example, tablet, syrup, or injection. Upon clicking the "Generate QR Code" button, a scannable QR image is produced that can be downloaded or shared with the doctor. Each QR code is linked to a unique URL (qr_url) stored in the Firestore database. This URL is intended to identify the associated medicine or prescription and can be used by other components of the system to access or verify relevant information. By generating and managing QR codes with URL links, the system supports more flexible patient verification and medication tracking, allowing administrators to assist with adherence monitoring or troubleshooting as needed.

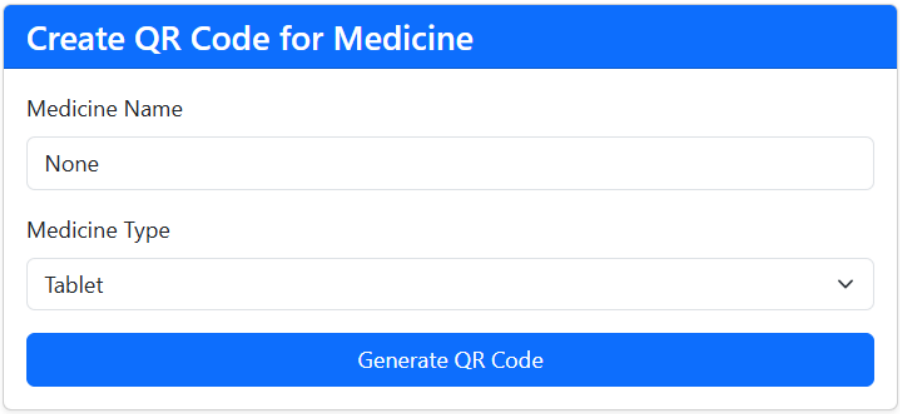


Figure 12. QR code generator

Figure 15 shows the Activity Log section of the My Medi Admin Dashboard. This feature records all key administrative actions performed within the system, including operations on patient accounts such as creation, update, password change, and deletion. Each entry in the log includes a precise timestamp, action type, object type (such as patient or admin), the relevant object ID, and details of the user who performed the action. For example, the log captures when a superadmin updates or deletes a patient record, providing a transparent record for auditing and review. At the top of the activity log, a "Clear All Activity Logs" button is visible. This action is restricted to superadmin users only; regular admins do not have permission to clear the log, further enforcing role-based access and protecting the integrity of system records. This comprehensive activity logging supports accountability, facilitates system monitoring, and assists in identifying potential unauthorized activities.

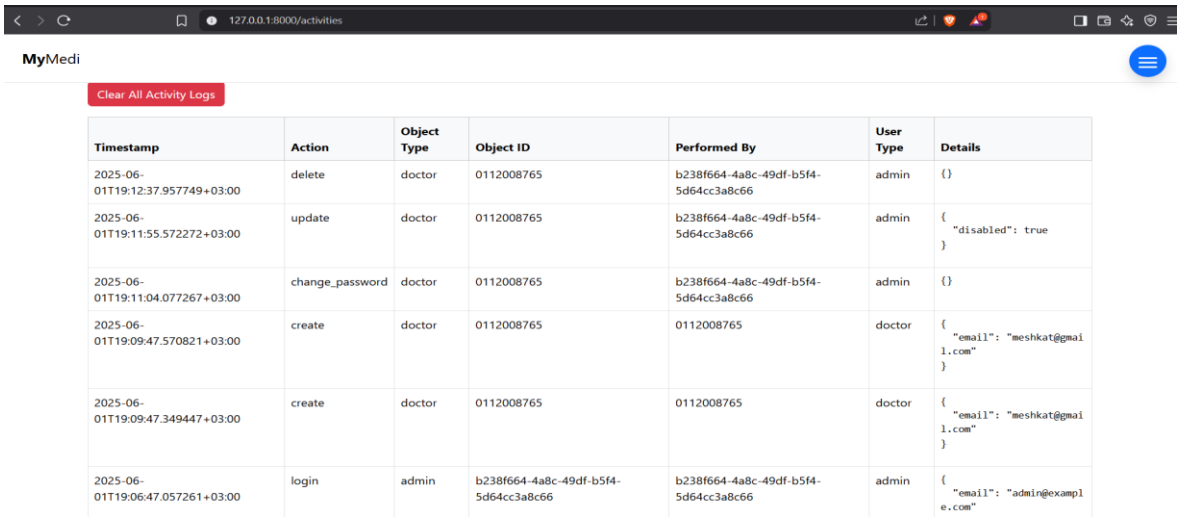


Figure 13. Activity log

The System Health page as per Figure 16, in the My Medi Admin Dashboard plays a key role in maintaining performance and ensuring stability within the system. As shown in Figure 16, this section visualizes real-time memory statistics of the backend server hosting the My Medi application. These metrics are collected using the psutil library in Python and provide insights into the server's total, available, used, and free memory, as well as the percentage of memory in use. By visualizing these statistics through a clear bar chart, administrators can easily assess the current load and system responsiveness. Additionally, the page includes a "Clear Old Data" button, which enables the admin to safely remove outdated records, such as inactive QR code entries or obsolete user activity logs. This feature helps maintain database efficiency and reduces potential memory bottlenecks without risking the integrity of important user data. Regular use of this tool ensures the platform remains optimized and reliable for continuous healthcare administration.

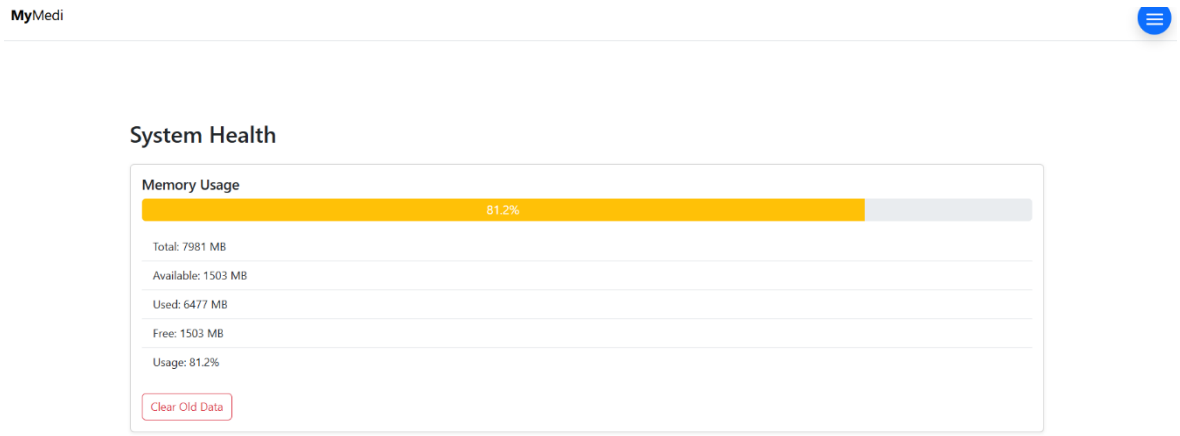


Figure 14 System health monitoring

To ensure the app is practical for hospital and government use, the UI uses large buttons, readable fonts, and colour-coded icons (e.g., red for delete, yellow for disable) to reduce mistakes. Each section loads quickly and was tested across screen sizes to ensure it works smoothly on tablets and laptops. In designing the admin panel, the focus was on clarity, core administrative capabilities, and role-based user management. The system seeks to make digital health infrastructure management accessible and user-friendly for institutions that may not have advanced technical staff.

4.2 Integration Requirements

Figure 17 illustrates the integration of the My Medi system, demonstrating how seamless communication is achieved between the mobile application (used by doctors and patients) and the web-based admin dashboard. In this architecture, Firebase acts as the shared backend for both platforms, enabling secure authentication, centralized user management, and real-time data exchange. This unified approach helps synchronize data and maintain security across all components of the platform.

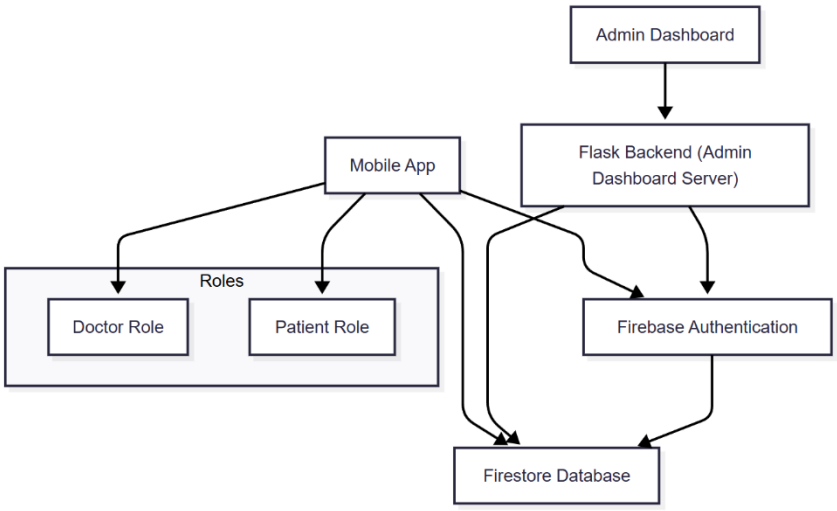


Figure 15. My Medi System Integration

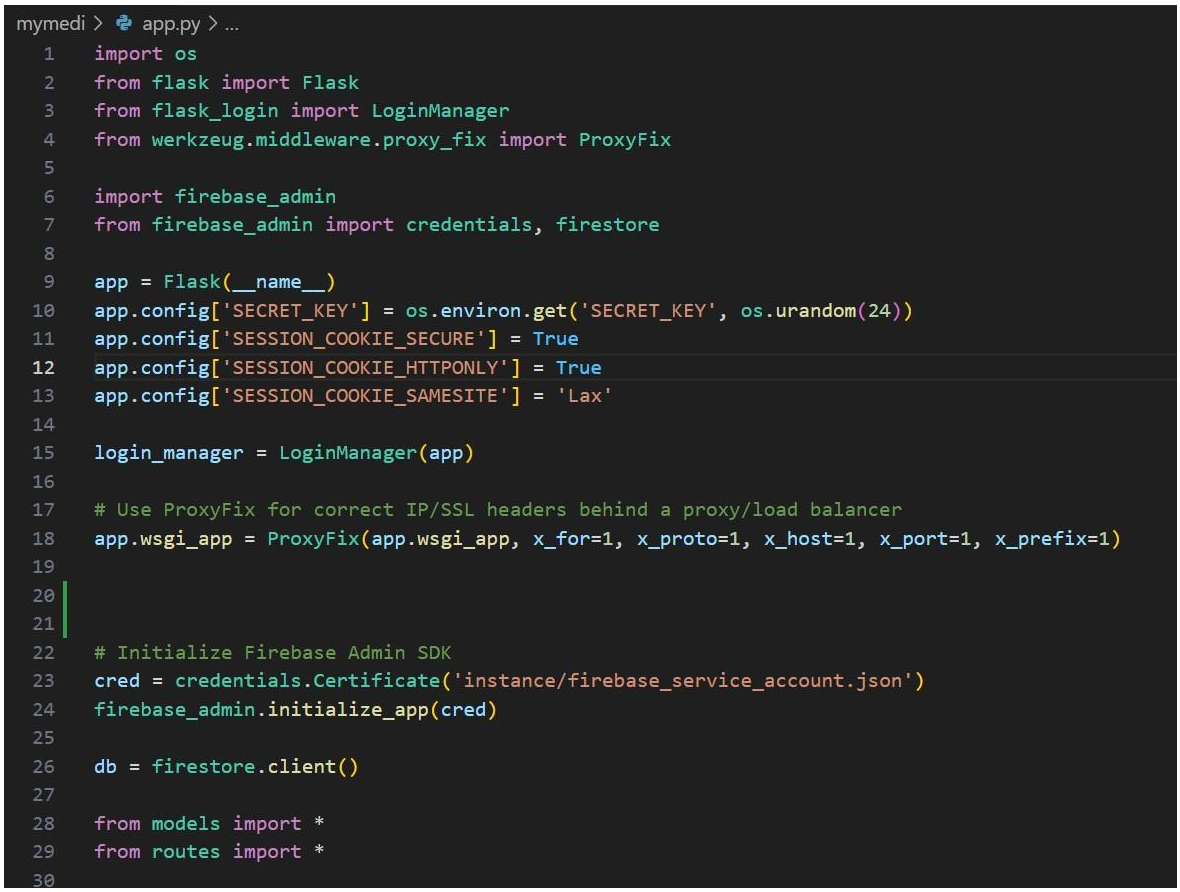
To maintain security and data separation, role-based login was implemented. Upon login, users were authenticated using Firebase Authentication, and their role (either doctor or patient) determined which portal and functionalities were accessible. Doctors were granted access to features such as prescription creation and QR code management within the mobile app, while patients were restricted to viewing medication reminders and confirming intake using QR scanning. Administrators, on the other hand, accessed the system via the web dashboard, where they managed users and oversaw the system's operations.

Integration was structured with clear role boundaries to ensure that medical data remained visible only to doctors and patients. Administrators were not given access to prescription details, thereby ensuring patient privacy and compliance with data protection laws such as GDPR (Saha 2023). Instead, their role focused on technical support such as assisting in user registration, resetting passwords, and generating QR codes upon request from doctors. The system architecture separates backend logic from client-side components to enhance security and maintainability. All administrative actions performed via the web-based admin dashboard are processed by the Flask backend, which communicates with Firebase services using the Firebase Admin SDK. The web client (browser) interacts exclusively with the Flask server and does not connect directly to Firebase or store any sensitive credentials. In contrast, the mobile application (for doctors and patients) connects directly to Firebase using client SDKs for real-time authentication and data access. This separation ensures that all sensitive operations are handled server-side for the admin panel, while the mobile app leverages Firebase's real-time features. This allows the system to leverage real-time updates for the mobile app while ensuring secure, controlled access through the admin dashboard.

4.3 Building the app

The My Medi Administration Web Application was developed using Python's Flask framework for backend logic and Firebase for user authentication and database services. The application was designed to be modular, secure, and user-friendly, enabling hospital or healthcare administrators to manage system infrastructure without accessing sensitive medical data. Emphasis was placed on real-time data handling and code clarity.

A modular folder structure was followed to enhance maintainability. The core of the application begins with the app.py file, presented in Figure 18, where the Flask instance is initialized. Important configurations, such as session security settings and cookie policies, were defined to protect user data during web interactions. The Firebase Admin SDK was initialized in this file to connect the app with Firebase services using a service account key. The Proxy Fix middleware was also included to ensure correct request headers behind a proxy. Additionally, blueprints and model imports were registered here to keep route definitions modular and organized.



```
mymedi > app.py > ...
import os
from flask import Flask
from flask_login import LoginManager
from werkzeug.middleware.proxy_fix import ProxyFix

import firebase_admin
from firebase_admin import credentials, firestore

app = Flask(__name__)
app.config['SECRET_KEY'] = os.environ.get('SECRET_KEY', os.urandom(24))
app.config['SESSION_COOKIE_SECURE'] = True
app.config['SESSION_COOKIE_HTTPONLY'] = True
app.config['SESSION_COOKIE_SAMESITE'] = 'Lax'

login_manager = LoginManager(app)

# Use ProxyFix for correct IP/SSL headers behind a proxy/load balancer
app.wsgi_app = ProxyFix(app.wsgi_app, x_for=1, x_proto=1, x_host=1, x_port=1, x_prefix=1)



# Initialize Firebase Admin SDK
cred = credentials.Certificate('instance/firebase_service_account.json')
firebase_admin.initialize_app(cred)

db = firestore.client()

from models import *
from routes import *

```

Figure 16. Flask setup and Firebase initialization

Routes.py file, which is shown in Figure 19, handled the main logic for routing and user template rendering. Routes were defined for login, logout, dashboard access, user creation, QR generation, and system health monitoring. Login functionality was implemented using Firebase Authentication, where the submitted email and password were verified. If valid, user roles were checked to grant access to admin-only sections. Flask-Login was used for session management, with the @login_required decorator ensuring that only authenticated users could access protected routes. Template filters were defined to format timestamps and dates before rendering them in the UI.

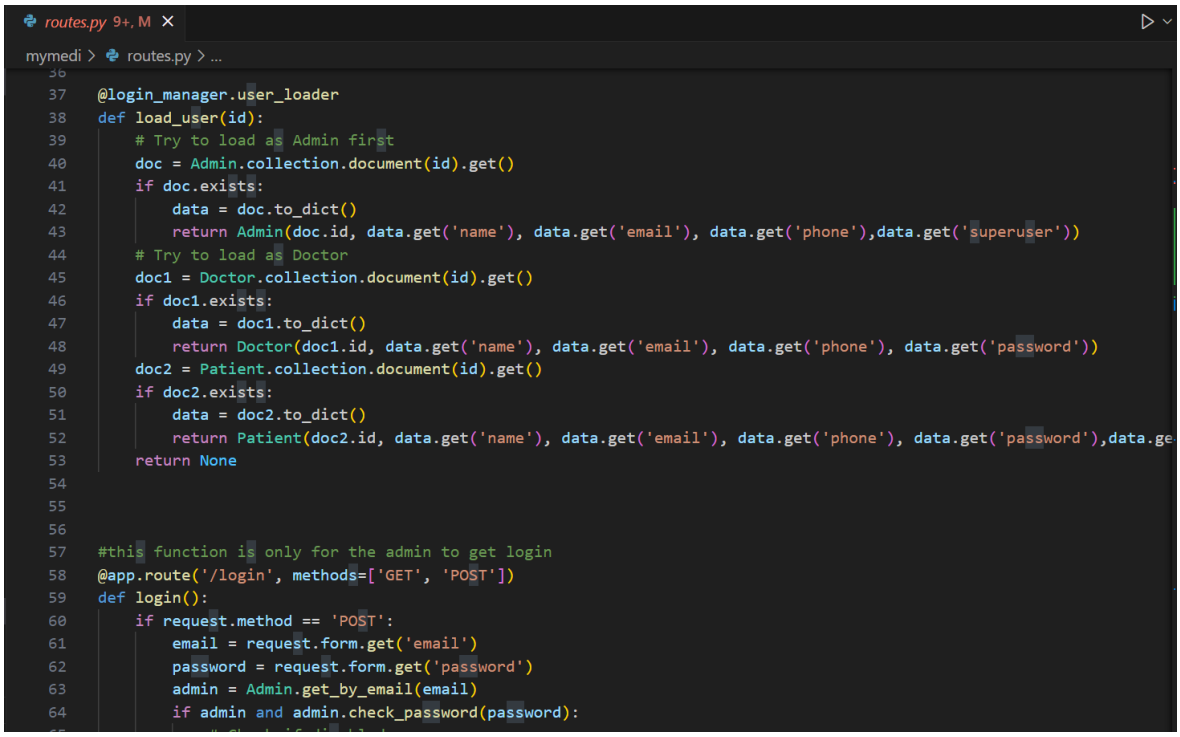


Figure 17. Session management and login routing

TEst_firebase.py file, which is shown in Figure 20, is used to enable user interaction with the Firestore database. A shared dB object was created from the Firebase Admin SDK, which was imported across multiple modules. This object allowed real-time read and write access to user records. The test_firebase.py script was developed to confirm Firebase connectivity by querying data from the 'admins' collection. If the connection failed, an error message was printed. This served as a diagnostic tool during initial development.

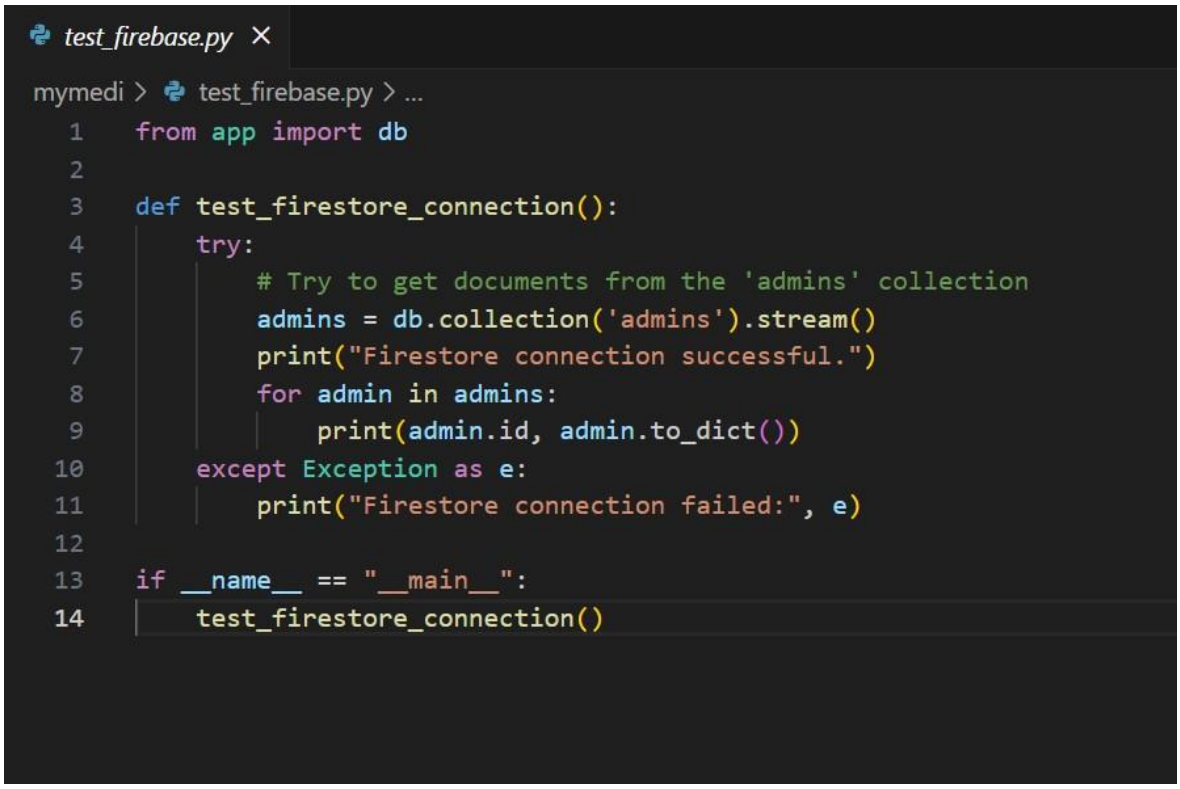
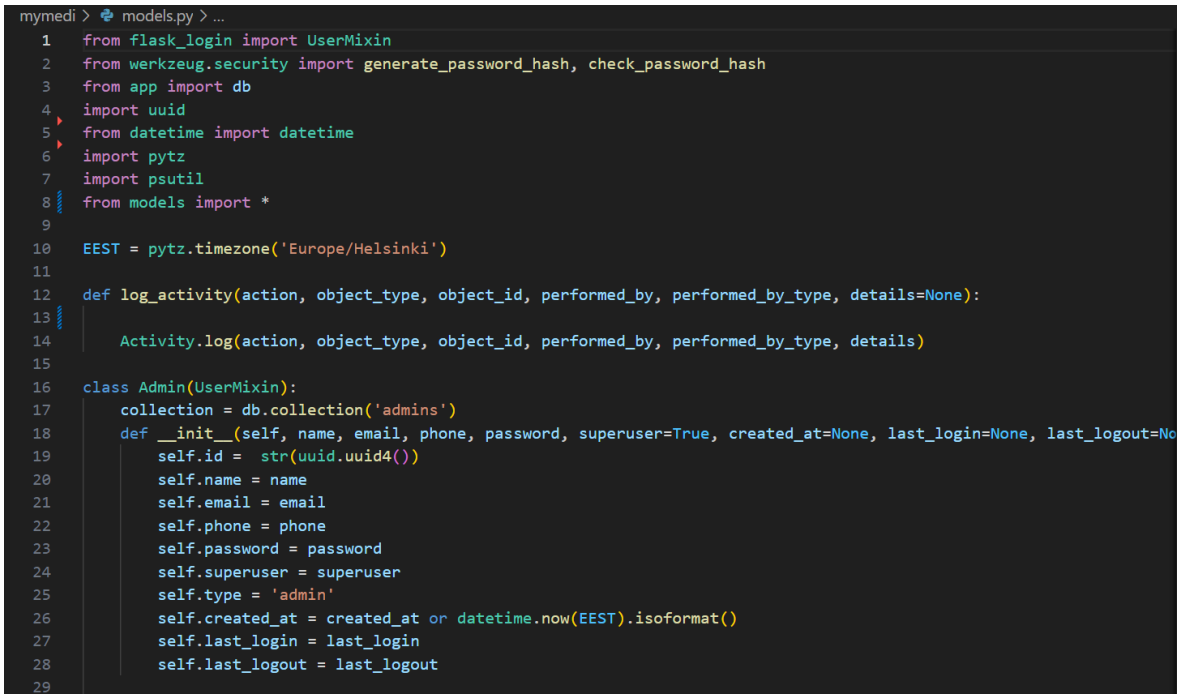


Figure 18. Firestore connection test

In Figure 21 the models.py file defined the data models and logic related to users. Classes like Admin, Doctor, and Patient were defined here, each mapping to a Firestore collection. These models included methods for setting attributes such as name, email, password (stored securely with hashing), timestamps for creation, login, and logout. A log_activity () function was also implemented to write activity logs into Firestore whenever an account was created, updated, or deleted.



```
mymedi > models.py > ...
1  from flask_login import UserMixin
2  from werkzeug.security import generate_password_hash, check_password_hash
3  from app import db
4  import uuid
5  from datetime import datetime
6  import pytz
7  import psutil
8  from models import *
9
10 EEST = pytz.timezone('Europe/Helsinki')
11
12 def log_activity(action, object_type, object_id, performed_by, performed_by_type, details=None):
13
14     Activity.log(action, object_type, object_id, performed_by, performed_by_type, details)
15
16 class Admin(UserMixin):
17     collection = db.collection('admins')
18     def __init__(self, name, email, phone, password, superuser=True, created_at=None, last_login=None, last_logout=No
19         self.id =  str(uuid.uuid4())
20         self.name = name
21         self.email = email
22         self.phone = phone
23         self.password = password
24         self.superuser = superuser
25         self.type = 'admin'
26         self.created_at = created_at or datetime.now(EEST).isoformat()
27         self.last_login = last_login
28         self.last_logout = last_logout
29
```

Figure 19. Admin model and logging function

To assist doctors facing technical issues with the mobile application, a QR code generator was implemented using the qrcode Python library shown in Figure 22. This functionality allowed the admin to manually create and distribute QR codes associated with patient prescriptions. These codes could be scanned via the mobile app by patients to confirm medication intake.

```python
class QRCode:
    collection = db.collection('qrcodes')
    def __init__(self, medicine_name, qr_url=None, id=None, created_at=None, medicine_types=None):
        self.id = id
        self.medicine_name = medicine_name
        self.qr_url = qr_url
        self.created_at = created_at or datetime.now(EEST).isoformat()
        self.medicine_types = medicine_types or []

    def generate(self):
        import qrcode as pyqrcode
        import io
        img = pyqrcode.make(self.medicine_name)
        buf = io.BytesIO()
        img.save(buf, format='PNG')
        buf.seek(0)
        return buf

    def save(self, performed_by=None, performed_by_type=None, action='create', details=None):
        # Save QRCode metadata to Firestore
        doc_ref = QRCode.collection.document()
        self.id = doc_ref.id
        doc_ref.set({
            'medicine_name': self.medicine_name,
            'qr_url': self.qr_url or '',
            'created_at': self.created_at,
            'medicine_types': self.medicine_types,
        })
```

Figure 20. QR code generation function

In Figure 23 system activity tracking was implemented using a lightweight logging utility, where most of the events, such as user creation or deletion, was recorded with a timestamp and performed-by metadata. These logs could be viewed from the admin dashboard to enhance transparency and ease of debugging.

```python
def log_activity(action, object_type, object_id, performed_by, performed_by_type, details=None):

    Activity.log(action, object_type, object_id, performed_by, performed_by_type, details)
```

Figure 21. Activity log

Furthermore, a System Health monitoring panel was included to track memory usage and clean up outdated or unnecessary data. This ensured that the application remained performant and stable even with long-term usage. While this function does not interact directly with user data, it supports the reliability of the application under administrative control.

In conclusion, the My Medi Admin Dashboard was developed as a secure, modular system using Flask and Firebase. The code were provided to illustrate critical parts of the backend logic, including server setup, Firebase connection, route handling, user model definitions, QR code generation, and system activity logging. Together, these components enabled seamless integration and support for the mobile app while ensuring data security, session control, and user management.

5 Testing and Results

5.1 Testing the app

Manual testing was conducted to observe the core features, user interactions, and access controls of the My Medi Admin Web Application. The following Figures illustrates specific test cases and their outcomes.

Doctor Registration and Feedback: When a new doctor was registered through the admin dashboard, the system responded with a green success message at the top of the screen (Figure 24). This indicated that the input form was accepted and processed, providing immediate feedback to the administrator.

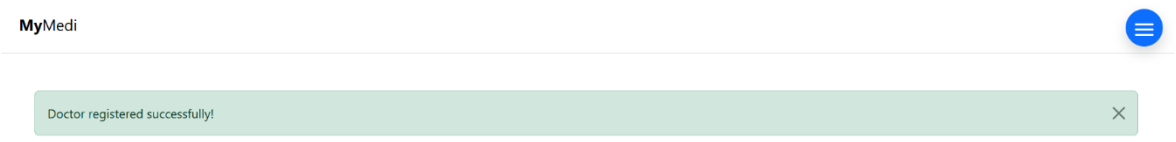


Figure 22. Doctor registered

Doctor Deletion Confirmation: After deleting a doctor account, the system displayed a similar confirmation banner stating, "Doctor deleted" (Figure 25). This confirmed that the delete action had been recognized and completed by the backend.

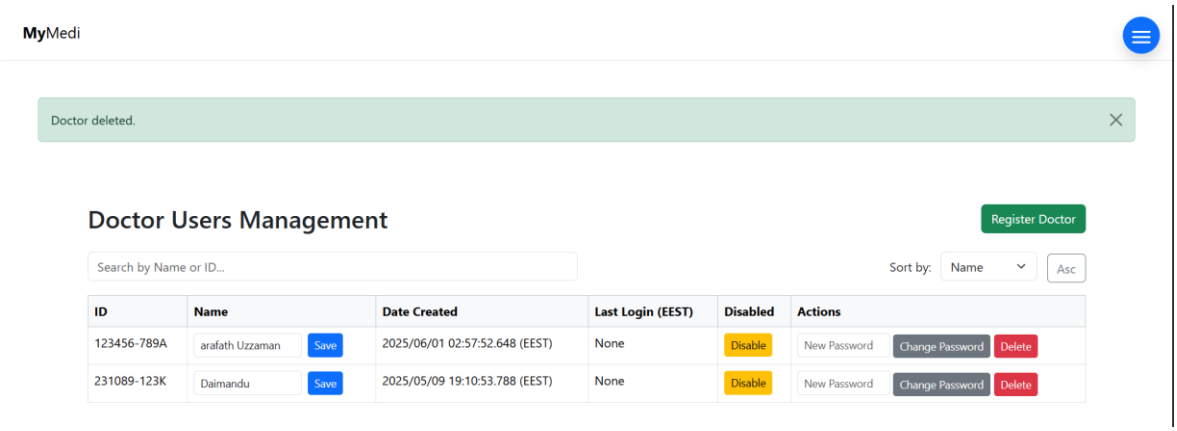


Figure 23. Doctor deletes

Doctor Account Disabled: To test disabling user accounts, a doctor profile was set to disabled via the dashboard. A message confirming the change was shown, and the UI updated to reflect the account's disabled status (Figure 26). This suggested that account state changes were being tracked and communicated clearly.

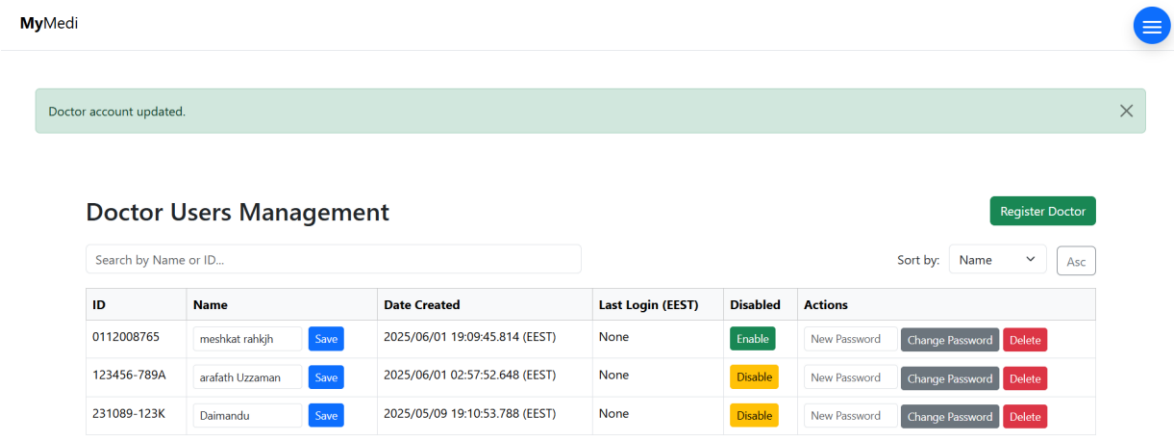


Figure 24. Doctor account Disable

Password Change Process: When a password was changed for a doctor account, a confirmation message appeared (Figure 27). This message verified that the password reset request was processed, which can be useful if users forget their login details.

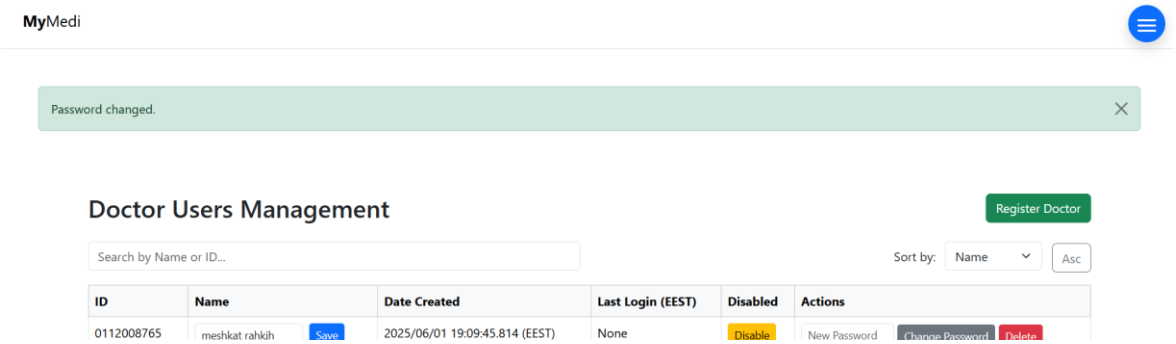


Figure 25. Doctor password change

Patient Management Testing: The same functional tests were performed for patient accounts. New patients were registered, existing records were edited, and accounts were disabled or deleted through the admin dashboard. Each action triggered clear feedback messages (similar to those shown in Figures 24–27), and all events were recorded in the activity log (see Figure 28). Password resets and login validation for patients were also tested, and the system responded with appropriate confirmation or error messages, confirming that patient-related workflows were working as expected.

Activity Log Functionality: The admin dashboard includes an activity log feature, as shown in Figure 28. This log recorded actions such as account creation, updates, deletions, and password changes, along with timestamps and user information. The activity log helps with basic auditing and system oversight.

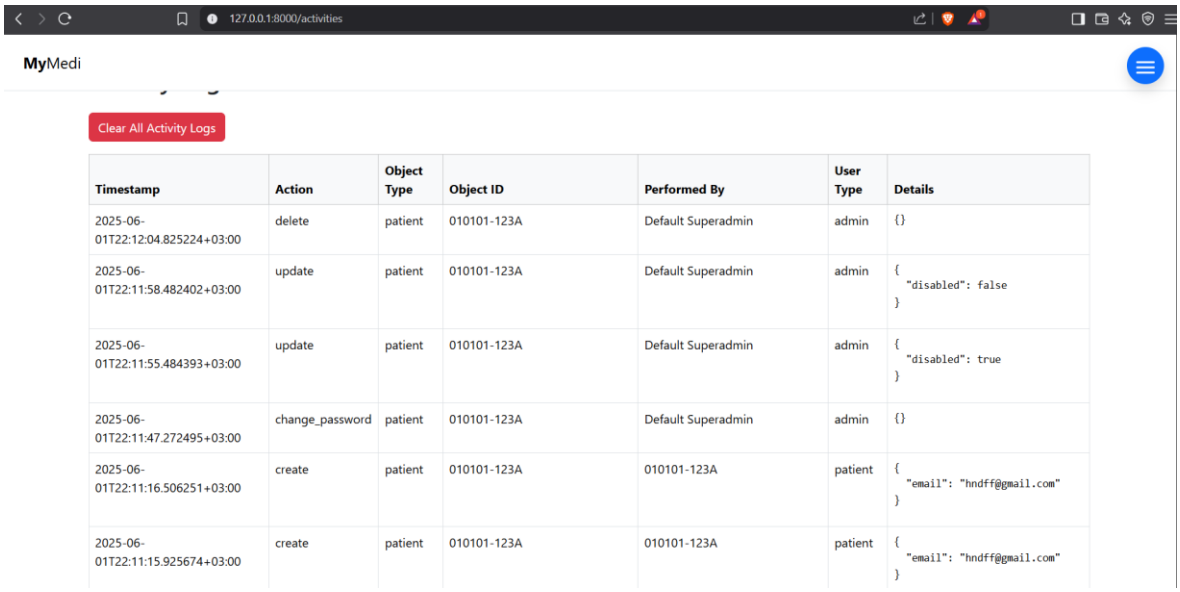


Figure 26. Activity log

Login Failure Handling: To examine authentication handling, an incorrect email and password combination was entered on the login screen. The application returned an "Invalid email or password" error message (Figure 29), indicating that invalid credentials were rejected and that user authentication checks are in place.

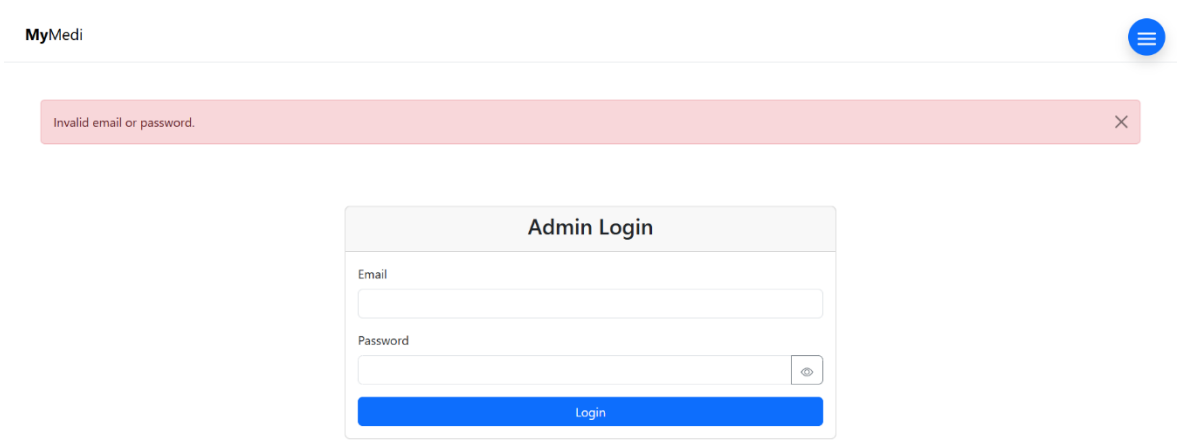


Figure 27. Invalid credentials

QR Code Deletion and Logging: Deleting a QR code from the system triggered a confirmation message, confirming successful deletion (Figure 30). This event was also recorded in the activity log, providing traceability for sensitive actions.

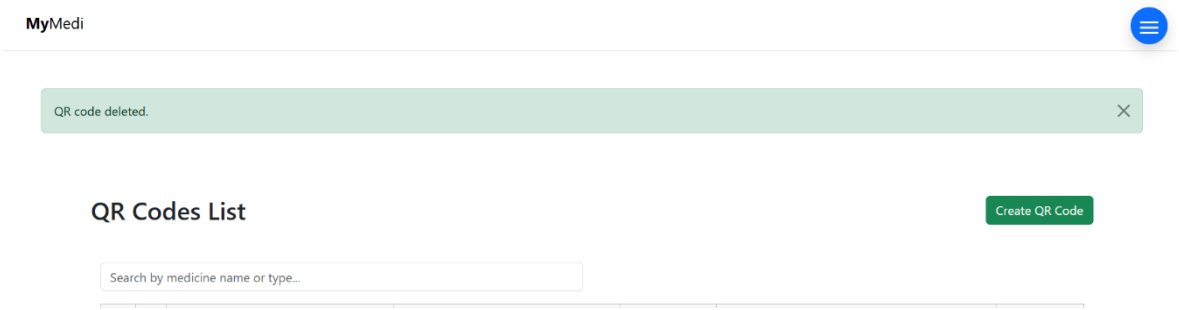


Figure 28. QR code delete

Role-Based Access Denial (Admin Section): An attempt to access a restricted admin page without appropriate permissions resulted in an "Access denied" message (Figure 31). This response demonstrated that access control mechanisms were present to block unauthorized users from sensitive sections.

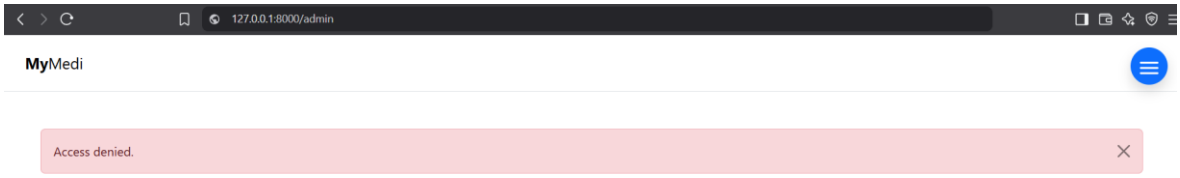


Figure 29. Admin access denied

Role-Based Access Denial (Superadmin Action): When a standard admin user tried to perform a superadmin-only action, the system displayed a message stating "Access denied. Only superadmin can perform this action" (Figure 32). This tested the system's enforcement of role-based privileges for higher-risk operations.

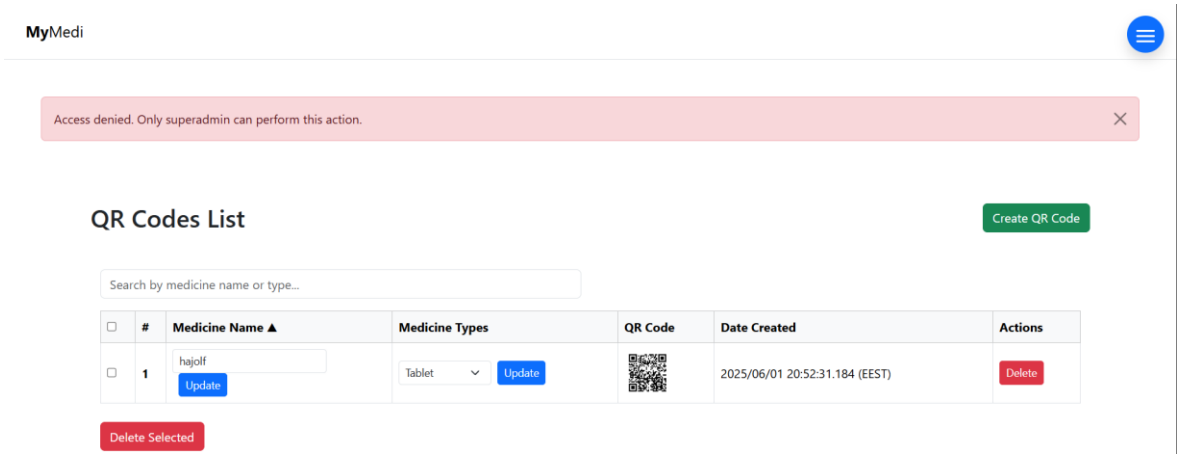


Figure 30. Access denied deleting QR code

Admin Registration Feedback: When registering a new admin user by the super admin, a confirmation message appeared (Figure 33). This provided immediate confirmation that the new account was created, supporting the system's administrative workflows.

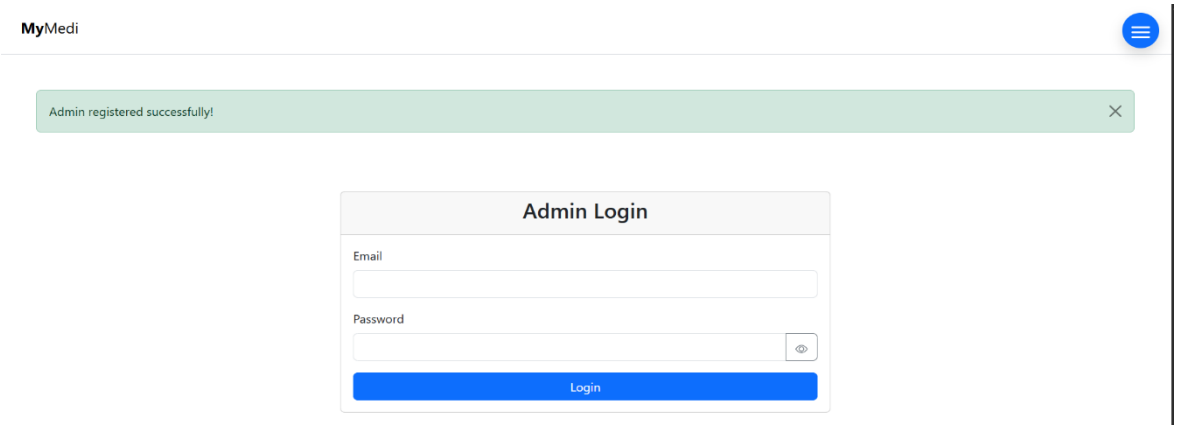


Figure 31. New admin created

5.2 Results

The My Medi Admin Web Application was subjected to several rounds of manual testing to verify its main features and to evaluate its reliability for healthcare administration. All tests were conducted in a simulated environment using test accounts rather than real hospital staff. Particular attention was given to the role-based access control system. During testing, both superadmin and regular admin accounts were created and used to confirm that permissions and restrictions functioned as intended. For example, only superadmins were able to register new admins, modify other admin accounts, or clear activity logs, while regular admins were restricted from performing these actions and received clear "access denied" messages (Figure 32). Admin users, on the other hand, were able to manage doctor and patient accounts, create and manage QR codes, and perform other permitted administrative tasks. All such operations triggered real-time updates in the Firebase backend, and the user admin panel provided immediate visual feedback for successful actions and errors.

The system activity log consistently recorded all major administrative events, including account creation, deletion, updates, and failed login attempts, thereby supporting traceability and security auditing (Figure 28). The layout and design were found to be intuitive and suitable for non-technical users in the test environment, with confirmation dialogs and color-coded alerts improving clarity during task execution. All the testing result are provided in Table 1, where the testing description and its outcome have been provided with the prior Figures. It should be noted that while the core functions and role-based restrictions were verified in testing, these tests were performed solely by developers and test users; no hospital personnel or actual patients were involved. Therefore, while the system's intended

behaviours were confirmed under test conditions, further validation in a real-world healthcare setting would be needed for comprehensive evaluation

Table 1. Test cases performed on the My Medi Admin Dashboard and their outcomes

Test Item	Description	Result	Figure Reference
Doctor Registration	Registering a new doctor via the admin dashboard	Pass	Figure 24
Doctor Editing/Disabling/Deleting	Editing, disabling, or deleting a doctor account	Pass	Figure 25, Figure 26
Doctor Password Change	Changing a doctor's account password	Pass	Figure 27
Patient Registration	Registering a new patient	Pass	Figure 28
Patient Edit-ing/Disabling/Deleting	Editing, disabling, or deleting a patient account	Pass	Figure 28
Patient Password Reset	Resetting patient password	Pass	Figure 28
Activity Log Recording	Logging all admin actions	Pass	Figure 28
Login Failure Handling	Entering incorrect email/password	Pass	Figure 29
QR Code Generation/Deletion	Generating and deleting QR codes	Pass	Figure 14, Figure 30
Role-Based Access	Admin denied access or action	Pass	Figure 32, Figure 33
Admin Registration	Registering new admin by superadmin	Pass	Figure 33
System Health Monitoring	Monitoring memory and clearing outdated data	Pass	Figure 16

In summary, the application functioned as expected in terms of core operations, security, and role-based access, and the admin panel proved usable within the scope of manual testing. Any claims about suitability for hospital environments are based only on these internal tests, and not on trials involving clinical staff or live patient data.

6 Conclusion and future development

This thesis detailed the development of the My Medi Admin Web Application as a proof of concept (PoC) platform, which supports medication management and compliance monitoring for elderly patients. Built using Flask and Firebase, the system enables secure user authentication, role-based admin privileges, QR code generation for prescriptions, and comprehensive activity logging. Manual testing confirmed that core modules user and prescription management, security features, and logging worked as intended. The system's layout was designed to be accessible for non-technical healthcare staff, and activity logs were implemented to provide transparency for all administrative actions. While manual tests attempted to assess usability for typical users, further evaluation with actual healthcare staff would be needed for comprehensive validation.

It was discovered during the development of the proof-of-concept application that two particular attributes internal communication and analytics capabilities would further enhance the potential value of a real-world system. First, the introduction of an integrated notification or alert system would improve communication between administrators and users (doctors and patients). Such a feature would allow for real-time messaging about account changes, system updates, or urgent issues directly within the application, reducing dependence on external channels and improving operational responsiveness. Second, expanding the analytics capabilities of the admin dashboard would enable administrators to visualize trends in user activity, monitor key performance indicators, and proactively address potential bottlenecks or security events. This would add significant value for data-driven decision-making and system maintenance.

In conclusion, the work emphasized the importance of robust security, privacy-conscious workflows, and role-based access control in administrative healthcare applications. By prioritizing future enhancements in internal communication and analytics, the platform will be better positioned to meet the evolving needs of healthcare organizations, offering greater transparency, coordination, and operational resilience.

References

Ashok Kumar, S. 2018. Mastering Firebase for Android Development: Build Real-Time, Scalable, and Cloud-enabled Android Apps with Firebase. Packt Publishing.

Banning, M. 2008. Older people and adherence with medication: a review of the literature. International Journal of Nursing Studies, 45(10), 1550–1561.

Barros, R. Jorge. 2014. Design and evaluation of a mobile user interface for older adults: navigation, interaction and visual design recommendations. Procedia Computer Science, 27, 369–378.

Cutler, R. L. Fernandez-Llimos, F. Frommer, M. Benrimoj, C. Garcia-Cardenas, V. 2018. Economic impact of medication non-adherence by disease groups: a systematic review. BMJ Open, 8(1). Accessed 28.5.2025. Available at: https://bmjopen.bmj.com/content/8/1/e016982

Dosecast Pill Reminder. 2023. Dosecast. Accessed 11.5.2023. Available at: https://dosecast.com/

Firebase. 2024. Firebase Security Rules. Accessed 5.3.2025. Available at: https://firebase.google.com/docs/rules

Flask Documentation. 2024. Routing. Accessed 3.3.2025. Available at: https://flask.palletsprojects.com/en/latest/quickstart/#routing

Flask-Login. 2024. User session management. Accessed 4.4.2025. Available at: https://flask-login.readthedocs.io/en/latest/

Flask-Security-Too. 2024. Security Features. Accessed 5.5.2025. Available at: https://flask-security-too.readthedocs.io/en/stable/

Gellad, W. F. Grenard, J. L. Marcum, Z. A. 2011. Systematic review of barriers to medication adherence in the elderly: looking beyond cost and regimen complexity. American Journal of Geriatric Pharmacotherapy, 9(1), 11–23.

Gaspar, D. Stouffer, J. 2018. Mastering Flask Web Development: Build enterprise-grade, scalable Python web applications, 2nd Edition. Packt Publishing Ltd.

GeeksforGeeks. 2025. Flask Tutorial. Accessed 12.5.2025. Available at: https://www.geeksforgeeks.org/flask-tutorial/

Google Firebase Documentation. 2024. Firebase Database REST API. Accessed 2.2.2025. Available at: https://firebase.google.com/docs/reference/rest/database/

Grinberg, M. 2018. Flask Web Development: Developing Web Applications with Python. O'Reilly Media, Inc.

Greenhalgh, T. Wherton, J. et al. 2017. Beyond adoption: a new framework for theorizing and evaluating nonadoption, abandonment, and challenges to the scale-up, spread, and sustainability of health and care technologies. Journal of Medical Internet Research, 19(11), e367.

Jormanainen, V. 2023. Large-scale implementation of the national Kanta services in Finland 2010–2018 with special focus on electronic prescription. Helsinki: University of Helsinki.

Medisafe. 2023. How Medisafe works. Accessed 1.12.2023. Available at: https://www.medisafe.com/

Moroney, L. 2017. The Definitive Guide to Firebase: Build Android Apps on Google's Mobile Platform. Apress.

MyTherapyApp. 2021. MyTherapy and accessibility. Accessed 3.11.2021. Available at: https://www.mytherapyapp.com/

Norton, P. C. Samuel, A. Aitel, S. Foster-Johnson, D. Richardson, E. Diamond, L. Parker, J. Roberts, M. 2005. Beginning Python. New York: Wiley & Sons, Inc.

Osterberg, L. Blaschke, T. 2005. Adherence to medication. New England Journal of Medicine, 353, 487–497.

Plone Conf, Boston. 2016. Python web framework from scratch. Accessed 17.10.2016. Available at: https://oz123.github.io/advanced-python/book/middlewares.html

Ronacher, A. 2024. Flask documentation: request lifecycle. Accessed 4.4.2024. Available at: https://flask.palletsprojects.com/en/latest/reqcontext/

Ronacher, A. 2024. Flask documentation: foreword. Accessed 3.3.2025. Available at: https://flask.palletsprojects.com/en/stable/en/latest/foreword

Saha, B. 2023. Analysis of the adherence of mHealth applications to HIPAA technical safeguards.

Takabi, H. Joshi, J. B. Ahn, G. 2010. Security and privacy challenges in cloud computing environments. IEEE Security & Privacy, 8(6), 24–31.

Thakkar, J. R. Laba, T. L. et al. 2016. Mobile telephone text messaging for medication adherence in chronic disease. JAMA Internal Medicine, 176(3), 340–349.

Wolford, B. 2018. What is GDPR, the EU's new data protection law. Accessed 25.5.2018. Available at: https://gdpr.eu/what-is-gdpr/

Yahiaoui, H. 2017. Firebase Cookbook. Packt Publishing.