

INTI International College Penang School of Engineering and Technology 3+0 Bachelor of Science (Hons) in Computer Science, in collaboration with Coventry University, UK 3+0 Bachelor of Science (Hons) in Computing, in collaboration with Coventry University, UK

Coursework cover sheet

Section A - To be completed by the student

Full Name: KISHEN KUMAAR A/L GANESH RAJA, VIKNES A/L THANGASPAREN, ARSYAD HASSAN BIN SEGU HASAN GANI, SATISH A/L PRAKASHAM	
CU Student ID Number: 14007673, 14008924, 13446824, 13459581	
Semester: AUGUST 2024	
Lecturer: R.K. KRISHNAMOORTHY Email: kmoorthy.ksamy@newinti.edu.my Extension: 345	
Module Code and Title: INT6005CEM SECURITY	
Assignment No. / Title: SECURE APPLICATION DEVELOPMENT	Assignment Type: Individual 50% of Module Mark
Hand out date: 30th September 2024	Due date: 15th November 2024
Penalties: NO Late work will be accepted. If you are unable to submit coursework on time due to extenuating circumstances, you may be eligible for an extension. Please consult the lecturer.	


Declaration: I/we the undersigned confirm that I/we have read and agree to abide by the University regulations on plagiarism and cheating and Faculty coursework policies and procedures. I/we confirm that this piece of work is my/our own. I/we consent to appropriate storage of our work for plagiarism checking.

Signature(s):

Group Member 1: *Kishen*

Name: KISHEN KUMAAR A/L GANESH RAJA

Student ID: 14007673

Group Member 2: 

Name: VIKNES A/L THANGASPAREN

Student ID: P23014982 / 14008924

Group Member 3: *ARSYAD*

Name: ARSYAD HASSAN BIN SEGU HASAN GANI

Student ID: P22014749 / 13446824

Group Member 4: *SATISH*

Name: SATISH A/L PRAKASHAM

Student ID: P22014510 / 13459581

Section B - To be completed by the module leader

Intended learning outcomes assessed by this work: 1. Develop and evaluate software that addresses the most common and most severe security concerns. 2. Critically evaluate the security of an IT ecosystem.		
Marking scheme	Max	Mark
1. Report <i>Refer to the section on Marking Scheme on detailed breakdown.</i> 2. System Functionality <i>Refer to the section on Marking Scheme on detailed breakdown.</i>	60 40	
Total	100	
<u>Lecturer's Feedback</u>		
<u>Internal Moderator's Feedback</u>		

Table Of Content

Chapter 1 : Introduction and System Overview.....	5
Chapter 2 : Potential Issues and Recommendations.....	6
2.1. Lack of back buttons in the app once logged in to the roles.....	6
2.2 Recommendations.....	7
Chapter 3 : Security Implementation.....	8
3.1 Encryption and Decryption of the system (Kishen-P23014943).....	8
3.1.1 The main screen and the roles of users.....	8
3.1.2 The encrypted password saved in firebase.....	9
3.1.3 The encrypted password saved in firebase been decrypted and shown in the system..	9
3.2.1 User Registration Input Validation Handling (Viknes, P23014982).....	10
3.2.2 Password Hashing for User Account (Viknes, P23014982).....	13
3.2.3 Data Privacy Policy (Viknes, P23014982).....	14
3.3.1 Error Handling (Satish P22014510).....	15
3.3.2 Database Error Handling in Registration Process (Satish P22014510).....	15
3.3.3 Database Error Handling in Patient Register Process (Satish P22014510).....	17
3.3.4 Database Error Handling in Doctor Registration Process (Satish P22014510).....	18
3.3.5 UI Error Handling (Satish P22014510).....	19
3.4 (Arsyad Hassan, P22014749).....	20
3.4.1 Role-Based Access Control (RBAC) and Session Management (Arsyad Hassan, P22014749).....	20
3.4.2 Session Management for Role-Based Access.....	21
3.4.3 Role-Specific Access in Main Views.....	25
3.4.4 Permissions-Based Access Control in Actions.....	26
3.4.5 Login and Registration for Each Role.....	28
3.4.6 Unauthorised Access Handling.....	31
Chapter 4 : Discussion.....	33
4.1 Key Findings.....	33
4.2 Future Enhancements of System.....	34
Chapter 5 : Conclusion.....	35
5.0 Conclusion and Learning Outcomes.....	35
References.....	36
Appendix.....	37

Chapter 1 : Introduction and System Overview

The "CallADoctor" Android application is designed to streamline the process of booking medical appointments, making healthcare more accessible and efficient. The application enables patients to place appointment requests, which doctors can review and either accept or decline based on availability.

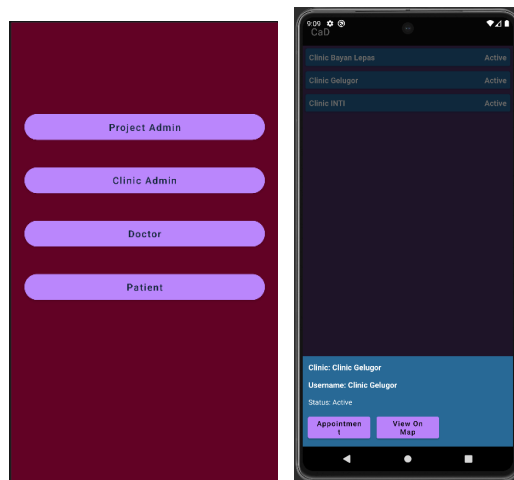


Figure 1.1 CallADoctor User Login Roles User Interface

Figure 1.2 Patient New Appointment Screen

The system employs Role-Based Access Control (RBAC) to handle 4 different user roles. The user roles consists of:

- Project Admin
- Clinic Admin
- Doctor
- Patient

This role based approach ensures data privacy of the authenticated users and also enhances the user experience by providing personalised functionalities for each role.

The CallADoctor app focuses on security and privacy, which is important when handling sensitive health-related data.

Chapter 2 : Potential Issues and Recommendations

2.1. Lack of back buttons in the app once logged in to the roles

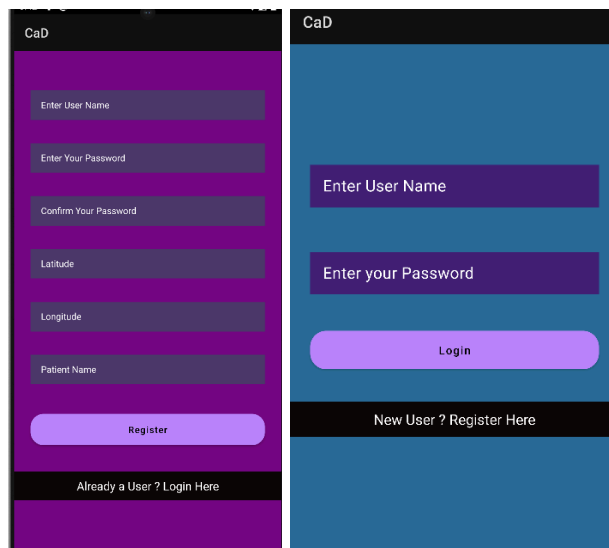


Figure 2.1 Back_Button issue

There is no back buttons to redirect us to the back page / previous page . Such as, administrator_SessionManager, Clinic_SessionManager, Doctor_SessionManager and patient_SessionManager.

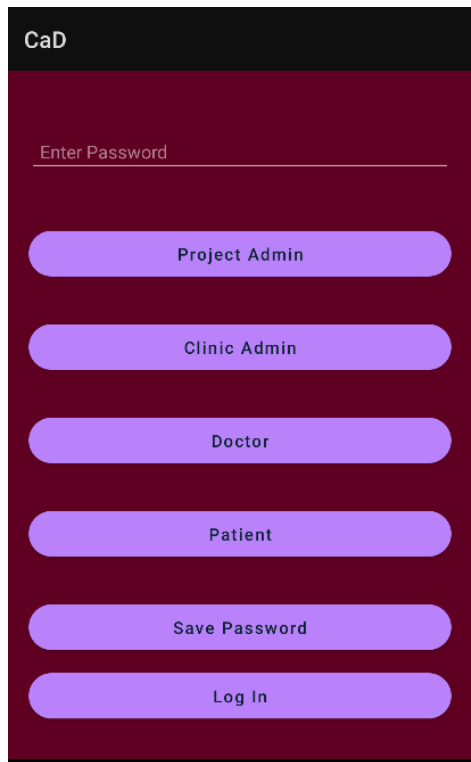


Figure 2.1.2 There is no proper LOGO for the app and UI design is simple

Figure 2.1.2 shows that the app doesn't have a proper logo and the UI design is simple

2.2 Recommendations

Based on the issues identified, the following suggestions should be helpful to improve the application;

1. Need to implement back buttons in order to have smooth flow of the app.
2. Implement better UI design, the colour for each roles so that easy to identify.
3. Implement extra user such as pharmacist so that user can directly order medication under that specific role.

Chapter 3 : Security Implementation

3.1 Encryption and Decryption of the system (Kishen-P23014943)

3.1.1 The main screen and the roles of users

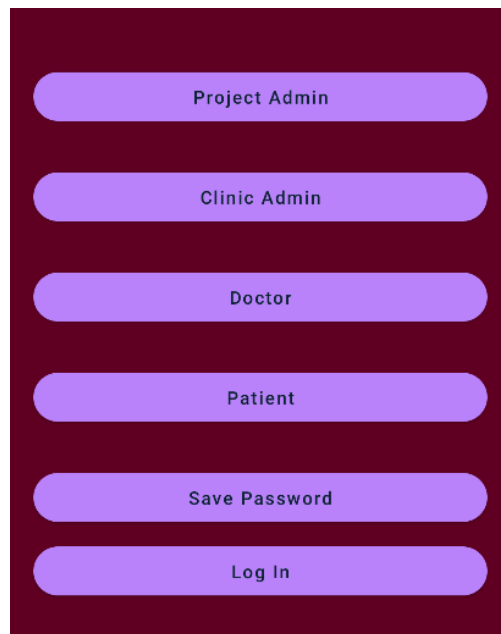
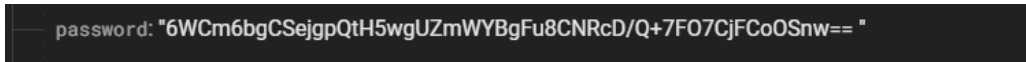


Figure 3.1.1 The main screen of the app

Figure 3.1.1 show the main screen and list of roles consist of Project_Admin, Clinic_Admin, DoctorLogin and Patient “Enter Password” field in each role will encrypt the password once the user has registered

3.1.2 The encrypted password saved in firebase

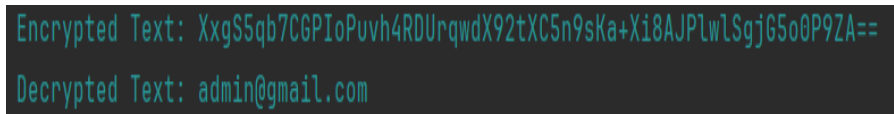
A screenshot of a dark-themed interface showing a label 'password:' followed by a long, complex string of characters in quotes: "6WCm6bgCSejgpQtH5wgUZmWYBgFu8CNRcD/Q+7F07CjFCoOSnw==".

```
password: "6WCm6bgCSejgpQtH5wgUZmWYBgFu8CNRcD/Q+7F07CjFCoOSnw=="
```

Figure 3.1.2 The encrypted password saved in firebase

Once the password has been encrypted, it will be saved in the firebase

3.1.3 The encrypted password saved in firebase been decrypted and shown in the system

A screenshot of a dark-themed interface showing two lines of text. The first line is 'Encrypted Text: XxgS5qb7CGPIoPuvh4RDUrqwdX92tXC5n9sKa+Xi8AJPlwLSgj65o0P9ZA==' and the second line is 'Decrypted Text: admin@gmail.com'.

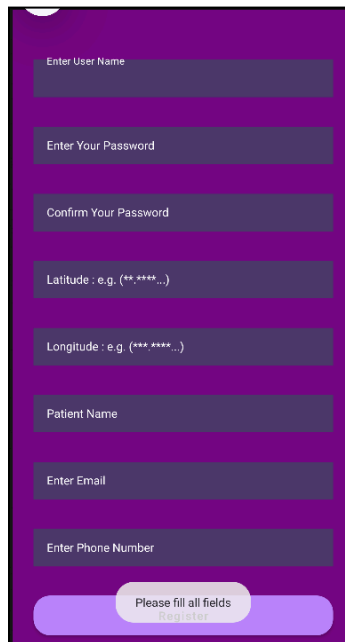
```
Encrypted Text: XxgS5qb7CGPIoPuvh4RDUrqwdX92tXC5n9sKa+Xi8AJPlwLSgj65o0P9ZA==  
Decrypted Text: admin@gmail.com
```

Figure 3.1.3 The encrypted password saved in firebase been decrypted and shown in the system

Figure 3.1.2 and figure 3.1.3 shows the encryption and decryption process of the system. and when the user logs in to the specific role in order to proceed to the dashboard of the selected role, the encrypted password will be decrypted from the firebase and will check whether it is the same or not to proceed with the login.

3.2 (Viknes, P23014982)

3.2.1 User Registration Input Validation Handling (Viknes, P23014982)



```
private boolean validateInputs(String patientName, String userName, String password, String confirmPassword, String email, String phone) {  
    if (confirmPassword.isEmpty() || email.isEmpty() || phone.isEmpty()) {  
        Toast.makeText(context, "Please fill all fields", Toast.LENGTH_SHORT).show();  
        return false;  
    }  
  
    if (patientName.isEmpty()) {
```

Figure 3.2.1 Input Validation for empty fields

Figure 3.2.2 Code Snippet for empty field validation

In the User Registration process, input validation ensures all required fields are filled before submission. If any fields (like patient name, username, password, etc.) are empty, the system displays an error message prompting the user to complete them. This validation prevents incomplete data and enhances the registration process.

Figure 3.2.1 shows the error message displayed when fields are empty, and **Figure 3.2.2** presents the code snippet for validating empty fields.

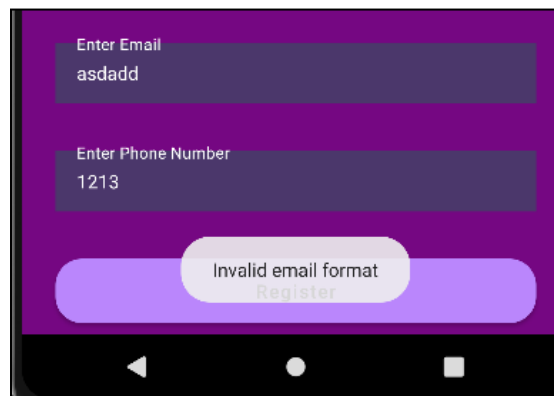
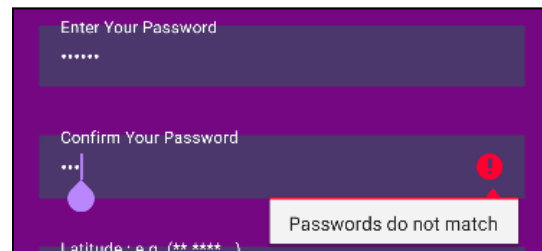
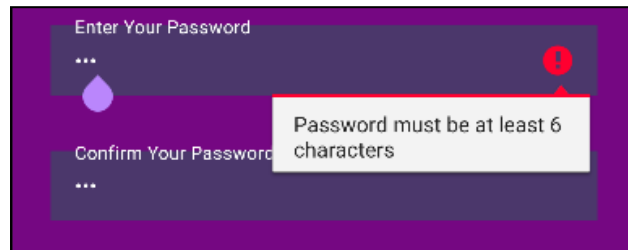


Figure 3.2.3 Password Minimum Characters

Figure 3.2.4 Confirm Password Validation

Figure 3.2.5 Wrong Email Input Format

The **figure 3.2.3** above demonstrates the validation for ensuring that the password is at least 6 characters long. The error message would appear when the user enters a password shorter than 6 characters.

The **figure 3.2.4** above displays the validation that checks if the password and confirm password fields match. It will show an error if the two fields do not match.

The **figure 3.2.5** demonstrates the validation for ensuring a correctly formatted email address. The error message will appear if the user enters an invalid email format.

Enter Phone Number

1213

Invalid phone number. Include country code.

Register

Latitude : e.g. (**.****...)

sadad

Invalid latitude

Longitude : e.g. (***.****...)

Latitude : e.g. (**.****...)

60.123

Longitude : e.g. (***.****...)

sdfsf

Invalid longitude

Patient Name

Figure 3.2.6 Phone Number Format Validation

Figure 3.2.7 Wrong Latitude Input Format

Figure 3.2.8 Wrong Longitude Input Format

The figure 3.2.6 demonstrates the validation to ensure that the phone number is correctly formatted, including the country code and containing only numeric characters. Next, the figure 3.2.7 displays the validation for the latitude field to ensure that the value is numeric and within the valid range of -90 to +90 degrees. Figure 3.2.8 demonstrates the validation for the longitude field to ensure that the input is numeric and within the range of -180 to +180 degrees.

3.2.2 Password Hashing for User Account (Viknes, P23014982)

- Register

```
//method to hash the password, using SHA-256 algorithm, before storing in the database.
private String hashPassword(String password) {
    try {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] hash = digest.digest(password.getBytes());
        StringBuilder hexString = new StringBuilder();
        for (byte b : hash) {
            String hex = Integer.toHexString(0xff & b);
            if (hex.length() == 1) hexString.append('0');
            hexString.append(hex);
        }
        return hexString.toString();
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

Figure 3.2.9 Method to Password hashing before storing in Database

Figure 3.2.9 shows the method used for hashing the password before storing it in the Firebase Database during user registration. This ensures that the actual password is not exposed in the database.

- Login

```
// method to convert the password entered, to make it into a hash (SHA-256),
// and then compare it with the stored hash, to Success Login.
private String hashPassword(String password) {
    try {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] hash = digest.digest(password.getBytes());
        StringBuilder hexString = new StringBuilder();
        for (byte b : hash) {
            String hex = Integer.toHexString(0xff & b);
            if (hex.length() == 1) hexString.append('0');
            hexString.append(hex);
        }
        return hexString.toString();
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

```
// Hash the entered password, call the method.
String hashedInputPassword = hashPassword(password);
if (storedPassword != null && storedPassword.equals(hashedInputPassword)) {

    patientSessionManager.createSession(userName, patientName);
    Toast.makeText(context, PatientLogin.this, "Welcome", Toast.LENGTH_SHORT).show();
    // starting a main activity.
    startActivity(new Intent(context, PatientMainView.class));

} else {
    loadingPB.setVisibility(View.INVISIBLE);
    Toast.makeText(context, PatientLogin.this, "Invalid Credential..", Toast.LENGTH_SHORT).show();
}
```

Figure 3.2.10 Method to Password hashing for login

Figure 3.2.11 Compare password entered with password stored

The above figure 3.2.10 demonstrates the method used to hash the password entered by the user during login.

Next, the figure 3.2.11 illustrates the comparison process where the hashed password entered by the user is compared with the previously stored hashed password in the database. If the hashes match, the login is successful.

3.2.3 Data Privacy Policy (Viknes, P23014982)

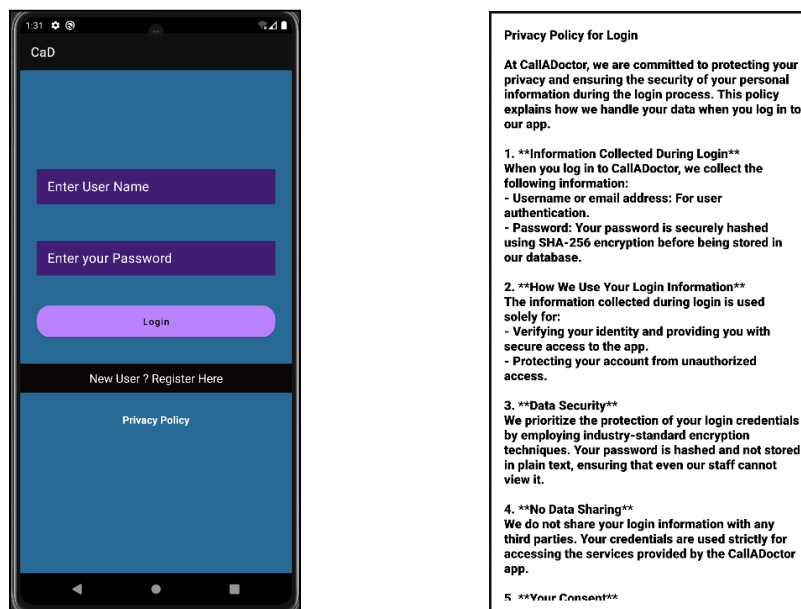


Figure 3.2.12 Patient Login Privacy Policy

Figure 3.2.13 Data Privacy Policy of CallADoctor

The figure in 3.2.12 displays the privacy policy notice presented to patients during the login process, ensuring they acknowledge the terms before accessing the application.

The 3.2.13 figure illustrates the full data privacy policy as presented in the application settings, accessible to all users for transparency and compliance.

3.3 (Satish P22014510)

3.3.1 Error Handling (Satish P22014510)

Error handling techniques are used in our program to guarantee stability, enhance user experience, and avoid crashes. Potential faults during application runtime can be identified and managed with the use of efficient error handling. This covers handling problems with network connectivity, erroneous user input, data access, and unforeseen system behaviours.

3.3.2 Database Error Handling in Registration Process (Satish P22014510)

Error handling is incorporated into the ClinicRegister activity to guarantee resilience and enhance the user experience throughout the registration procedure. Managing database-related problems that could arise when adding a new clinic to the Firebase database is the main goal of the error handling strategy.

```
// Database reference to add clinic
DatabaseReference clinicsReference = databaseReference;
clinicsReference.orderByChild( path: "clinicName").equalTo(ClinicName).addListenerForSingleValueEvent(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot snapshot) {
        if (!snapshot.exists()) {
            clinicsReference.orderByChild( path: "userName").equalTo(UserName).addListenerForSingleValueEvent(new ValueEventListener() {
                @Override
                public void onDataChange(@NonNull DataSnapshot snapshot) {
                    if (!snapshot.exists()) {
                        databaseReference.child(ClinicName).setValue(clinicModel)
                            .addOnSuccessListener(aVoid -> {
                                Toast.makeText( context: ClinicRegister.this, text: "Clinic Registered Successfully", Toast.LENGTH_SHORT).show();
                                startActivity(new Intent( packageContext: ClinicRegister.this, ClinicLogin.class));
                            })
                            .addOnFailureListener(e -> {
                                loadingPB.setVisibility(View.INVISIBLE);
                                Toast.makeText( context: ClinicRegister.this, text: "Registration failed. Try again.", Toast.LENGTH_SHORT).show();
                            });
                    } else {
                        loadingPB.setVisibility(View.INVISIBLE);
                        Toast.makeText( context: ClinicRegister.this, text: "Username already exists.", Toast.LENGTH_SHORT).show();
                    }
                }
            });
        }
    }
});
```

Figure 3.3.1 addOnSuccessListener and addOnFailureListener code snippet

```

        @Override
        public void onCancelled(@NonNull DatabaseError error) {
            loadingPB.setVisibility(View.INVISIBLE);
            Toast.makeText(context: ClinicRegister.this, text: "Error: " + error.getMessage(), Toast.LENGTH_SHORT).show();
        }
    });
} else {
    loadingPB.setVisibility(View.INVISIBLE);
    Toast.makeText(context: ClinicRegister.this, text: "Clinic name already exists.", Toast.LENGTH_SHORT).show();
}
}

@Override
public void onCancelled(@NonNull DatabaseError error) {
    loadingPB.setVisibility(View.INVISIBLE);
    Toast.makeText(context: ClinicRegister.this, text: "Error: " + error.getMessage(), Toast.LENGTH_SHORT).show();
}
}
});

```

Figure 3.3.2 onCancelled code snippet

First, the application handles database insertions during registration by utilising `addOnSuccessListener` and `addOnFailureListener`. The user is alerted and taken to the login page if the clinic's data is successfully saved. The progress bar disappears and a failure notice asks the user to try again if there is an insertion error—which could be caused by network or database problems.

Second, the software uses database queries to look for clinic names and usernames that already exist in order to avoid duplicate entries. Every query has an `onCancelled` callback that provides instant feedback by hiding the progress bar and displaying an error message in the event that the query fails.

3.3.3 Database Error Handling in Patient Register Process (Satish P22014510)

```
DatabaseReference patientsReference = databaseReference;
patientsReference.orderByChild( path: "userName").equalTo(UserName.replaceAll( regex: "[@.#$]", replacement: "-")).addListener
@Override
public void onDataChange(@NonNull DataSnapshot snapshot) {
    if (!snapshot.exists()) {
        databaseReference.child(UserName.replaceAll( regex: "[@.#$]", replacement: "-")).setValue(patientModel)
        .addOnSuccessListener(aVoid -> {
            hideLoading(); // Hide progress bar on success
            Toast.makeText( context: PatientRegister.this, text: "Patient Registered Successfully", Toast.LENGTH
            finish(); // End the activity
        })
        .addOnFailureListener(e -> {
            hideLoading(); // Hide progress bar on failure
            Toast.makeText( context: PatientRegister.this, text: "Registration failed. Try again.", Toast.LENGTH
        });
    } else {
        hideLoading(); // Hide progress bar if username exists
        Toast.makeText( context: PatientRegister.this, text: "Username already exists.", Toast.LENGTH_SHORT).show();
    }
}

@Override
public void onCancelled(@NonNull DatabaseError error) {
    hideLoading(); // Hide progress bar on query cancellation
    Toast.makeText( context: PatientRegister.this, text: "Error: " + error.getMessage(), Toast.LENGTH_SHORT).show();
}
});
```

Figure 3.3.3 Code Snippet for Database Error Handling

First, whereas `addOnFailureListener` handles failures by concealing the progress bar and asking the user to try again, `addOnSuccessListener` verifies successful registration during data insertion by displaying a message and terminating the activity. Second, database searches look for existing entries to avoid duplicate usernames. In order to give the user unambiguous feedback in the event that a query fails, the `onCancelled` callback conceals the progress bar and displays an error message. Because of these safeguards, the registration procedure is reliable and easy to use.

3.3.4 Database Error Handling in Doctor Registration Process (Satish P22014510)

```
DatabaseReference doctorsReference = databaseReference;
doctorsReference.orderByChild( path: "doctorName").equalTo(DoctorName).addListenerForSingleValueEvent(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot snapshot) {
        if (!snapshot.exists()) {
            doctorsReference.orderByChild( path: "userName").equalTo(UserName).addListenerForSingleValueEvent(new ValueEventListener() {
                @Override
                public void onDataChange(@NonNull DataSnapshot snapshot) {
                    if (!snapshot.exists()) {
                        databaseReference.child(DoctorName).setValue(doctorModel)
                            .addOnSuccessListener(aVoid -> {
                                Toast.makeText( context: ClinicDoctorRegister.this, text: "Doctor Added Successfully", Toast.LENGTH_SHORT).show();
                                finish();
                            })
                            .addOnFailureListener(e -> {
                                loadingPB.setVisibility(View.INVISIBLE);
                                Toast.makeText( context: ClinicDoctorRegister.this, text: "Failed to add Doctor. Try again", Toast.LENGTH_SHORT).show();
                            });
                    } else {
                        loadingPB.setVisibility(View.INVISIBLE);
                        Toast.makeText( context: ClinicDoctorRegister.this, text: "Username already exists.", Toast.LENGTH_SHORT).show();
                    }
                }
            });
        } else {
            loadingPB.setVisibility(View.INVISIBLE);
            Toast.makeText( context: ClinicDoctorRegister.this, text: "Error: " + error.getMessage(), Toast.LENGTH_SHORT).show();
        }
    }
});
```

Figure 3.3.4 Database Error Handling for Doctor Registration

Firstly, during data insertion, `addOnSuccessListener` confirms a successful registration by displaying a message and finishing the activity. In case of a failure, such as network issues or database constraints, `addOnFailureListener` hides the progress bar and informs the user with an error message, prompting them to try again.

Secondly, to prevent duplicate entries, database queries are used to check for existing doctor names and usernames. If a query fails, the `onCancelled` callback hides the progress bar and displays an error message using `error.getMessage()` to provide immediate feedback to the user.

3.3.5 UI Error Handling (Satish P22014510)

```
// Show the loading progress bar
private void showLoading() { 1 usage
    loadingPB.setVisibility(View.VISIBLE);
    Register.setEnabled(false); // Disable the Register button to prevent multiple clicks
}

// Hide the loading progress bar
private void hideLoading() { 4 usages
    loadingPB.setVisibility(View.GONE);
    Register.setEnabled(true); // Re-enable the Register button
}
}
```

Figure 3.3.5 Code Snippet for UI Error Handling

The Patient Registration process includes effective UI error handling to improve user experience and ensure the interface remains responsive during operations.

Firstly, dedicated loading management methods are used to handle UI state changes consistently. The `showLoading()` method displays the progress bar while disabling the Register button to prevent duplicate submissions, while `hideLoading()` hides the progress bar and re-enables the button once the operation completes.

Secondly, improved user feedback ensures users are informed during long-running operations. The progress bar (loadingPB) provides a clear indication that the registration process is ongoing, preventing confusion or repeated actions.

Lastly, the consistent management of UI states across success, failure, and cancellation scenarios ensures the application behaves predictably, enhancing reliability and user satisfaction. These measures create a seamless, user-friendly registration experience.

3.4 (Arsyad Hassan, P22014749)

3.4.1 Role-Based Access Control (RBAC) and Session Management (Arsyad Hassan, P22014749)

In our application, Role-Based Access Control (RBAC) is implemented to manage access based on predefined roles: Administrator, Clinic, Doctor, and Patient. This RBAC structure enforces security by limiting each role's access to the relevant parts of the application, ensuring that users can only perform actions permitted for their specific role.

The RBAC model in this application is based on four main components:

Permissions: Includes all permissions available within the system, such as "view_clinics," "manage_appointments," and "edit_patient_records."

Roles: Lists the various roles, including Administrator, Clinic, Doctor, and Patient.

Role_Permission: Maps each role to a set of permissions, allowing each role to perform specific actions.

Users: Associates each user with a specific role, ensuring they can only access data and actions allowed for that role.

These tables allow flexible management of permissions by role, enabling clear distinctions between the actions that each user role can perform.

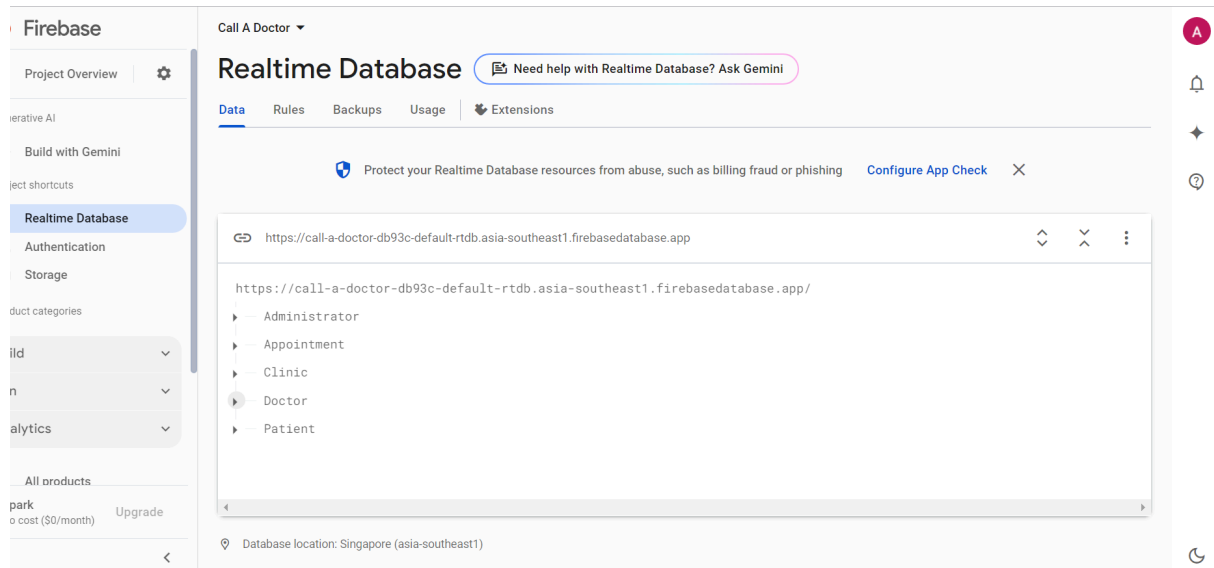


Figure 3.4.1: Firebase Database Structure for Role-Based Access Control

This figure shows the structure of the Firebase Realtime Database used for managing roles and permissions within the application. Nodes for Administrator, Appointment, Clinic, Doctor, and Patient are visible, each containing relevant data for the respective roles.

3.4.2 Session Management for Role-Based Access

Each role has a dedicated session manager that handles session creation, login verification, and secure redirection based on role. This ensures that each user type can only access the relevant sections of the application.

- Administrator_SessionManager manages sessions for administrators, verifying their credentials and storing their session securely.
- Clinic_SessionManager manages sessions for clinic users, allowing clinics to access clinic-specific functions.

- Doctor_SessionManager manages sessions for doctors, restricting access to doctor-specific functionalities.
- Patient_SessionManager manages sessions for patients, allowing only patients to access appointment and clinic details.

Each session manager includes key methods such as createSession, checkLogin, and logout to securely manage each role's session.

```

1 usage
public void createSession(String id) {
    editor.putBoolean(LOGIN, true);
    editor.putString(ID, EncryptionUtils.encrypt(id)); // Encrypt ID before storing
    editor.apply();
}

1 usage
public boolean isloggedin() { return sharedPreferences.getBoolean(LOGIN, defValue: false); }

public void checkLogin() {
    if (!this.isloggedin()) {
        // User is not logged in, redirect to Login Activity
        Intent i = new Intent(_context, AdministratorLogin.class);
        i.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.FLAG_ACTIVITY_CLEAR_TASK | Intent.FLAG_ACTIVITY_NEW_TASK);
        _context.startActivity(i);
    }
}

```

Figure 3.4.2: Administrator Session Manager - createSession and checkLogin Methods

This code snippet demonstrates the createSession and checkLogin methods in Administrator_SessionManager. The createSession method securely stores login information, and checkLogin verifies if the user is logged in, redirecting them to the login page if not.

```

public void createSession(String id, String clinicName){
    editor.putBoolean(LOGIN, true);
    editor.putString(ID, id);
    editor.putString(c_name, clinicName);
    editor.apply();
}

1 usage
public boolean islogin() { return sharedPreferences.getBoolean(LOGIN, defValue: false); }

public void checkLogin(){
    if(!this.islogin()){
        // user is not logged in redirect him to Login Activity
        Intent i = new Intent(_context, ClinicLogin.class);
        // Closing all the Activities
        i.addFlags(i.FLAG_ACTIVITY_CLEAR_TOP | i.FLAG_ACTIVITY_CLEAR_TASK | i.FLAG_ACTIVITY_NEW_TASK);
        // Starting Login Activity
        _context.startActivity(i);
    }
}

```

Figure 3.4.3: Clinic Session Manager - createSession and checkLogin Methods

This snippet shows the createSession and checkLogin methods in Clinic_SessionManager. The createSession method stores the clinic's session data, including clinic name, while checkLogin enforces access control by verifying the session status.

```

public void createSession(String id, String doc){
    editor.putBoolean(LOGIN, true);
    editor.putString(ID, id);
    editor.putString(DOCTOR, doc);
    editor.apply();
}

1 usage
public boolean isloggin() { return sharedPreferences.getBoolean(LOGIN, defValue: false); }

public void checkLogin(){
    if(!this.isloggin()){
        // user is not logged in redirect him to Login Activity
        Intent i = new Intent(_context, DoctorLogin.class);
        // Closing all the Activities
        i.addFlags(i.FLAG_ACTIVITY_CLEAR_TOP | i.FLAG_ACTIVITY_CLEAR_TASK | i.FLAG_ACTIVITY_NEW_TASK);
        // Staring Login Activity
        _context.startActivity(i);
    }
}

```

Figure 3.4.4: Doctor Session Manager - createSession and checkLogin Methods

This figure displays the createSession and checkLogin methods in Doctor_SessionManager. The session includes the doctor's ID and name, ensuring that only authenticated doctors can access the application's doctor-specific features

```

public void createSession(String id, String patientName){
    editor.putBoolean(LOGIN, true);
    editor.putString(ID, id);
    editor.putString(PATIENT, patientName);
    editor.apply();
}

1 usage
public boolean isloggin() { return sharedPreferences.getBoolean(LOGIN, defValue: false); }

public void checkLogin(){
    if(!this.isloggin()){
        // user is not logged in redirect him to Login Activity
        Intent i = new Intent(_context, PatientLogin.class);
        // Closing all the Activities
        i.addFlags(i.FLAG_ACTIVITY_CLEAR_TOP | i.FLAG_ACTIVITY_CLEAR_TASK | i.FLAG_ACTIVITY_NEW_TASK);
        // Staring Login Activity
        _context.startActivity(i);
    }
}

```


Figure 3.4.5: Patient Session Manager - createSession and checkLogin Methods

The createSession and checkLogin methods in Patient_SessionManager are shown here. These methods handle session storage and verification for patients, allowing access only to those with an active session.

3.4.3 Role-Specific Access in Main Views

Each main view for a role is protected by its respective session manager, ensuring that only authenticated users with the correct role can access it.

- AdministratorClinicView: Accessible only to administrators, allowing them to manage clinics and view clinic-related data.
- DoctorMainView: Accessible only to doctors, providing them with tools to view and manage appointments and prescriptions.
- PatientClinicView: Accessible only to patients, enabling them to view clinic details and book appointments.
-

Each view includes role-based session verification. For example, in AdministratorClinicView, the checkLogin method from Administrator_SessionManager ensures that only logged-in administrators can access the view:

```
administrator_sessionManager = new Administrator_SessionManager(getApplicationContext());  
administrator_sessionManager.checkLogin();
```

Figure 3.4.6: Administrator Login - Session Initialization and Access Control

This snippet demonstrates how an administrator's login credentials are verified in Firebase, followed by session creation. If the credentials are valid, a session is created, and the administrator is redirected to `AdministratorClinicView`.

In addition, role-specific data and functionality are limited to each main view. For instance, in `DoctorMainView`, the `getAppointment` method retrieves only the appointments relevant to the logged-in doctor.

3.4.4 Permissions-Based Access Control in Actions

Role-based permissions are also applied to specific actions within each main view to further restrict access. Although the current application does not have a separate permissions table, the logic is embedded in the action controls within each view.

For example, in `DoctorMainView`, the prescription button (`editBtn`) is only displayed if the patient record has no prescription, allowing doctors to add new prescriptions while preventing them from editing existing ones:

```

if(appModel.getPrescription().equals(""))
{
    editBtn.setText("Prescription");
}
else
{
    editBtn.setVisibility(View.GONE);
}

```

Figure 3.4.7: Clinic Login - Session Verification and Access Control

This code shows the clinic login verification process. After verifying credentials and the clinic's active status, a session is created, granting access to clinic-specific views and features.

```

if(clinicModel.getActive()==true)
{
    statusTV.setText("Status: Active");
}
else
{
    statusTV.setText("Status: Inactive");
}

```

Figure 3.4.8: Prescription Button Control for Doctor Role

This figure displays the conditional rendering of the "Prescription" button in DoctorMainView. The button only appears if no prescription is present, allowing doctors

to add new prescriptions and hiding the option if one already exists.

3.4.5 Login and Registration for Each Role

The application features unique login and registration activities for each role:

- AdministratorLogin: Verifies administrator credentials and initiates an administrator session.
- ClinicLogin: Authenticates clinic users, allowing access to clinic-specific functions upon login.
- DoctorLogin: Authenticates doctors, redirecting them to DoctorMainView for appointment and patient management.
- PatientLogin: Authenticates patients, allowing them to view clinics and request appointments.

```

adminRef.addListenerForSingleValueEvent(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot snapshot) {
        String storedUsername = snapshot.child( path: "username").getValue(String.class);
        String storedPassword = snapshot.child( path: "password").getValue(String.class);

        if (storedUsername != null && storedPassword != null) {
            if (storedUsername.equals(userName) && storedPassword.equals(passWord)) {
                administrator_sessionManager.createSession(userName);
                Toast.makeText( context: AdministratorLogin.this, text: "Login Successful..", Toast.LENGTH_SHORT).show();
                // starting a main activity.
                startActivity(new Intent( packageContext: AdministratorLogin.this, AdministratorClinicView.class));
            } else {
                loadingPB.setVisibility(view.GONE);
                Toast.makeText( context: AdministratorLogin.this, text: "Invalid Credential..", Toast.LENGTH_SHORT).show();
            }
        } else {
            loadingPB.setVisibility(view.GONE);
            Toast.makeText( context: AdministratorLogin.this, text: "Invalid Credential..", Toast.LENGTH_SHORT).show();
        }
    }
}

```

Figure 3.4.9: AdministratorLogin - Firebase Verification and Session Initialization

This code snippet shows the AdministratorLogin process, where administrator credentials are validated against Firebase. Upon successful verification, an administrator session is initiated, granting access to administrator-specific features in the application.

```

Login.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        loadingPB.setVisibility(View.VISIBLE);

        String userName = username.getText().toString();
        String passWord = password.getText().toString();

        DatabaseReference clinicsReference = databaseReference;
        clinicsReference.orderByChild( path: "userName").equalTo(userName).addListenerForSingleValueEvent(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot snapshot) {
                if(snapshot.exists())
                {
                    for (DataSnapshot userSnapshot : snapshot.getChildren()) {
                        String storedPassword = userSnapshot.child( path: "password").getValue(String.class);
                        boolean isActive = userSnapshot.child( path: "active").getValue(Boolean.class);
                        String clinicName = userSnapshot.child( path: "clinicName").getValue(String.class);

                        if (storedPassword != null && storedPassword.equals(passWord) && isActive) {

```

Figure 3.4.10: ClinicLogin - Authentication and Role-Based Session Creation

In this figure, the ClinicLogin process is shown, where the application verifies clinic user credentials and active status using Firebase. If authentication succeeds, a session for the clinic role is created, allowing access to clinic-specific functionalities within the app.

```

DatabaseReference clinicsReference = databaseReference;
clinicsReference.orderByChild( path: "userName").equalTo(userName).addListenerForSingleValueEvent(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot snapshot) {
        if(snapshot.exists())
        {
            for (DataSnapshot userSnapshot : snapshot.getChildren()) {
                String storedPassword = userSnapshot.child( path: "password").getValue(String.class);
                String doctorName = userSnapshot.child( path: "doctorName").getValue(String.class);

                if (storedPassword != null && storedPassword.equals(passWord)) {

                    doctorSessionManager.createSession(userName, doctorName);
                    Toast.makeText( context: DoctorLogin.this, text: "Welcome "+doctorName, Toast.LENGTH_SHORT).show();
                    // starting a main activity.
                    startActivity(new Intent( packageContext: DoctorLogin.this, DoctorMainView.class));

                } else {
                    loadingPB.setVisibility(view.INVISIBLE);
                    Toast.makeText( context: DoctorLogin.this, text: "Invalid Credential..", Toast.LENGTH_SHORT).show();
                }
            }
        }
    }
}

```

Figure 3.4.11: DoctorLogin - Authentication and Role-Based Redirection

This figure illustrates the DoctorLogin process, where doctor credentials are validated, and upon successful authentication, a session is created specifically for doctors. The doctor is then redirected to DoctorMainView, where they can manage appointments and patient information.

```
@Override
public void onClick(View view) {
    loadingPB.setVisibility(View.VISIBLE);

    String userName = username.getText().toString();
    String passWord = password.getText().toString();

    DatabaseReference clinicsReference = databaseReference;
    clinicsReference.orderByChild( path: "userName").equalTo(userName).addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot snapshot) {
            if(snapshot.exists())
            {
                for (DataSnapshot userSnapshot : snapshot.getChildren()) {
                    String storedPassword = userSnapshot.child( path: "password").getValue(String.class);
                    String patientName = userSnapshot.child( path: "patientName").getValue(String.class);

                    if (storedPassword != null && storedPassword.equals(passWord)) {

                        patientSessionManager.createSession(userName,patientName);
                        Toast.makeText( context: PatientLogin.this, text: "Welcome", Toast.LENGTH_SHORT).show();
                        // starting a main activity.
                        startActivity(new Intent( packageContext: PatientLogin.this, PatientMainView.class));
                    }
                }
            }
        }
    });
}
```

Figure 3.4.12: PatientLogin - Authentication and Access to Patient-Specific Features

This figure demonstrates the PatientLogin process. Firebase validates the patient's credentials, and if successful, a session is created for the patient. This login grants the patient access to view clinic information and request appointments.

3.4.6 Unauthorised Access Handling

When a user without the necessary permissions tries to access restricted functionality, the system prevents access and redirects them to the appropriate screen. For instance, in each role-specific view, session verification (checkLogin) ensures

unauthorised users are redirected to the login screen if they try to access protected areas.

The application displays error messages for unauthorised actions, preventing users from performing tasks outside their permission scope.

```
public void checkLogin() {  
    if (!this.isloggedin()) {  
        // User is not logged in, redirect to Login Activity  
        Intent i = new Intent(_context, AdministratorLogin.class);  
        i.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.FLAG_ACTIVITY_CLEAR_TASK | Intent.FLAG_ACTIVITY_NEW_TASK);  
        _context.startActivity(i);  
    }  
}
```

Figure 3.4.13: Administrator Session Verification with checkLogin Method

This code shows the checkLogin method in Administrator_SessionManager. If the administrator is not logged in, they are redirected to AdministratorLogin, ensuring only authenticated administrators access the application's main view.

Chapter 4 : Discussion

4.1 Key Findings

The assessment of the "CallADoctor" application highlights several significant findings in terms of security and user protection. Firstly, implementing input validation and password encryption mitigates risks related to incorrect email formats, weak passwords, and improper input types (like latitude and longitude), thereby enhancing the data integrity of user-provided information.

The application leverages Firebase for backend services, including authentication, database management, and data storage. Firebase's real-time database provides secure data storage and access, and with Firebase Authentication, each user's identity is verified before accessing protected resources. Passwords are securely stored using Firebase's authentication mechanisms, which include built-in support for hashing and encryption. This setup not only simplifies secure authentication but also reduces the risk of unauthorized access.

Session management within the app further enforces secure user access by utilizing individual session managers for each role (Administrator, Clinic, Doctor, and Patient). This ensures that users can only access the areas and functionalities specific to their role, minimizing the risk of unauthorized access to sensitive information. The Role-Based Access Control (RBAC) system integrated within the application segregates permissions effectively, defining specific actions and data access according to predefined roles. This role-specific access restriction not only improves data privacy but also strengthens data security by limiting user actions to only what their role permits.

Error handling has been thoughtfully implemented throughout the application, particularly in login and form submission processes. Instead of revealing system-specific details, error messages are user-friendly, enhancing the user experience while keeping system details secure. By leveraging Firebase's

error-handling capabilities, the app captures and manages errors related to authentication and data access effectively, ensuring a seamless experience even in cases of input or connectivity issues.

The combination of Firebase's security features, session management, RBAC, and comprehensive error handling contributes to a robust security infrastructure within "CallADoctor." These elements work together to safeguard sensitive user data and promote a trustworthy, reliable platform for users seeking healthcare services.

4.2 Future Enhancements of System

To further improve the security and usability of the "CallADoctor" application, several enhancements are recommended. Multi-factor authentication (MFA) could be implemented for critical roles, such as Administrator and Clinic users, to add an extra layer of security against unauthorised access.

Session management could be strengthened by adding session expiration and automatic logout features, reducing the risk of unauthorised access on shared devices. Expanding Role-Based Access Control (RBAC) with more granular permissions would allow for finer control over data access as the app scales.

Enhanced encryption using Firebase's support for HTTPS and SSL/TLS upgrades would improve data security in transit, while data access logging could enable monitoring and audits, helping to identify suspicious activities early on.

Lastly, clearer error handling with detailed HTTP status codes and custom messages would improve user experience, while routine security audits and Firebase analytics could ensure proactive detection and mitigation of vulnerabilities. These improvements would make "CallADoctor" more secure, scalable, and user-friendly.

Chapter 5 : Conclusion

5.0 Conclusion and Learning Outcomes

In conclusion, the security evaluation of the "CallADoctor" application highlights its strong commitment to user protection and data integrity. Key findings emphasise critical measures, such as input validation, session management, role-based access control (RBAC), and secure data handling via Firebase. These features collectively enhance the application's security posture, ensuring that sensitive information is protected and that access is restricted based on user roles.

Potential future enhancements, including continuous monitoring, multi-factor authentication (MFA), and SSL/TLS upgrades, demonstrate the app's adaptability to emerging security challenges. The emphasis on user-friendly error handling and structured access control reinforces both security and usability, fostering a reliable user experience.

Overall, this project provided valuable insights into the current strengths and areas for improvement, offering an opportunity to deepen understanding of secure application development. This experience fosters continuous learning and adaptation to cybersecurity standards, preparing for future challenges in building secure and resilient applications.

References

National Institute of Standards and Technology (NIST). (2000). Role-Based Access Control (RBAC) Standard. Gaithersburg, MD: NIST. Available at: <https://csrc.nist.gov/publications/detail/nist-standards/rbac/final> [Accessed 15 Nov. 2023].

Gollmann, D. (2011). Computer Security. 3rd ed. Chichester: Wiley.

Stallings, W. (2017). Network Security Essentials: Applications and Standards. 6th ed. Boston: Pearson.

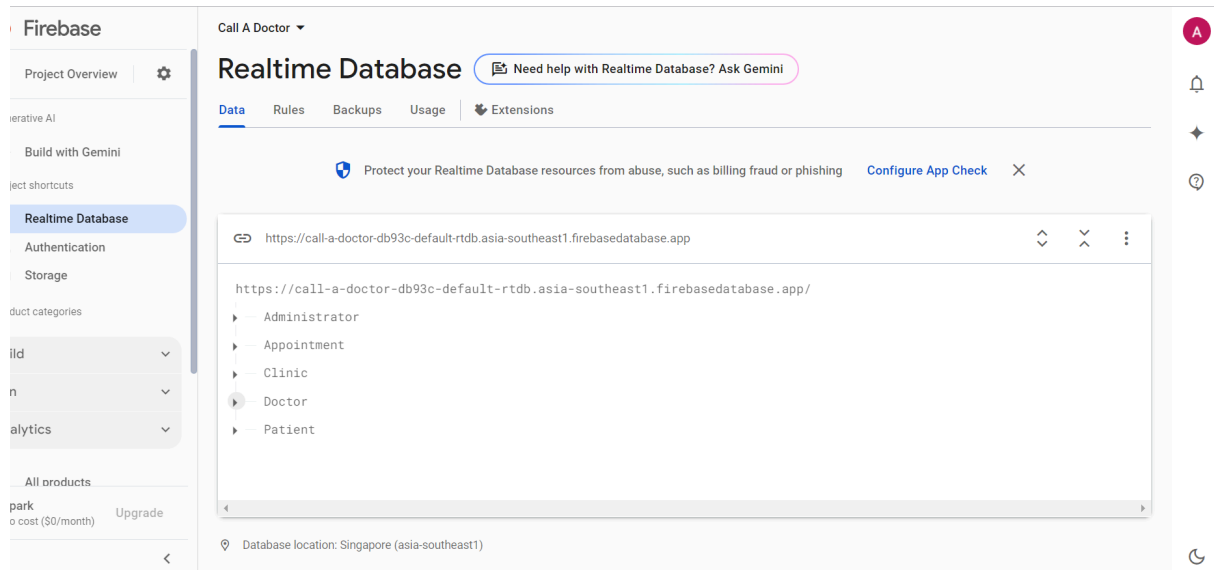
Chapman, P. (2019). 'Understanding Role-Based Access Control in Modern Applications'. International Journal of Information Security, 18(3), pp. 203-217.

Ilott, M. (2023). Password Hashing and Storage Basics. [online] Medium. Available at: <https://markilott.medium.com/password-storage-basics-2aa9e1586f98>.

MartinJ (2021). 4.1 Getting Professional with Firebase V9 - 'System Hygiene' - Error-handling and Transactions. [online] DEV Community. Available at: <https://dev.to/mjoycemilburn/41-getting-professional-with-firebase-v9-system-hygiene-error-handling-and-transactions-36jf>

Marcher, D. (2023). Effective Error Handling in Android: Strategies and Best Practices. [online] Medium. Available at: <https://blog.stackademic.com/effective-error-handling-in-android-strategies-and-best-practices-c9eb6fb4b864>.

Appendix



Firestore Implementation for this application

Turnitin Report

Group_14_6005CEM.pdf			
ORIGINALITY REPORT			
2%	1%	0%	2%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS
PRIMARY SOURCES			
1	Submitted to INTI Universal Holdings SDM BHD Student Paper	1%	
2	Submitted to University of Central Lancashire Student Paper	<1%	
3	Submitted to Griffith College Dublin Student Paper	<1%	
4	www.rroj.com Internet Source	<1%	
Exclude quotes On Exclude bibliography On Exclude matches < 10 words			

Github link

<https://github.com/KishenKumaarexe/CallaDoctor.git>