identifAi – User Manual

Group 19

Makris, Andreas (cdxc13)

Marquez, Aelyssa (kwsd37)

Meier, Andreas M. (qnzj81)

Wong, Titus Y.S. (hldl68)

Tolley, Lara M. (kwcq95)

Contents

1 Overview

identifAi is a mobile application created for *IBM*, designed for Android devices. The main purpose of this app is to allow users to discover information about other users around them using an AR environment and as a result, make it easier to form connections with new people who may have similar interests. This app also aims to benefit employers by finding people who may be interested in working for their company as well as easily provide their background information, such as any experience that may be advantageous to the company.

The final product should allow a user to create a personal profile made up of their professional and social interests. This profile is then accessible to nearby users within an AR environment by pointing the device camera at a user: their information will then appear around them on the screen in AR.

In the development of the app, *Unity* is the main platform used, within which C# scripts are written for the functionality of the app. The *ARFoundation* package is utilised for the AR environment, which includes *ARCore*, allowing development for Android devices. This package makes available device tracking and plane detection, which enables the app to track the device's location and orientation in actual space. This project also requires the *Firebase Unity* SDK to handle the backend environment. The Firebase products used for the app include *Firebase Authentication* for controlling user accounts and keeping passwords secure, and *Firestore*, a cloud database for storing users' data.

This document covers the prerequisites in Section 2 for continuing app development in a suitable environment on a computer or laptop, and also for downloading the app as a user on an Android phone. Each function of the app is described in Section 3, providing information for how to navigate the app as a user, as well as explanations for the corresponding C# code. Section 4 gives the structure of the database created in Firestore and how to manage the backend environment in Firebase. Finally, the last sections describe how to maintain the project and any future developments that can be implemented.

2 Prerequisites

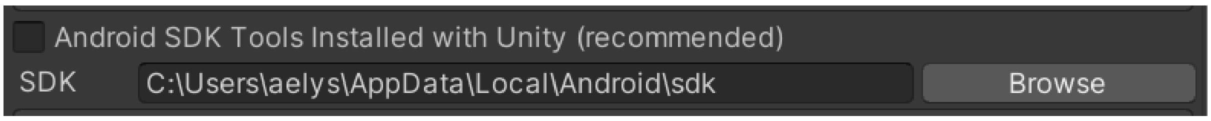
2.1 Software Requirements

Since the *identifAi* app is initially designed for Android phones (with the eventual possibility for expansion to iOS), currently the prerequisites for running the app are for Android devices. The app is compatible with Android phones with OS at or above *7.0 Nougat* and requires at least 50MB available storage space.

The following software requirements are for app development on a laptop or computer:

The most current stable release of the framework is *Unity Test Framework* 1.1.30, and the version of Unity used for this project is 2020.3.22f1 with Android Build Support. The version of Unity can be downloaded through *Unity Hub* (download) to open the project folder, provided in Section 2.2. These requirements were specified to support the use of the AR package, ARCore, which operates on the minimum API level 24, while setting it the lowest necessary level possible in order to accommodate more users. Instructions for setting up the relevant AR packages is also provided in Section 2.2.

The target API level is set at (SDK) level 32 to meet Google Play's developer requirements - corresponding to Android's beta version 12 (12L) to be compatible for Android's future releases. The Android SDK must be installed separately to Unity on the local machine and can be downloaded through the website either with *Android Studio* or solely through the basic Android command line tools. In Unity, the path to the Android SDK must be set manually: In the Unity bar at the top, go to Edit -> Preferences -> External Tools. Under Android, untick 'Android SDK Tools Installed with Unity' (shown in Figure 2.1) and set the SDK path to the location of the downloaded SDK.



Android SDK Tools Installed with Unity (recommended)

SDK C:\Users\aelys\AppData\Local\Android\sdk

Figure 2.1: SDK path location

The app is built in Unity using *Gradle* 6.9.1, which also must be installed on the local machine and can be downloaded through the website. The path to this version of Gradle must also be set manually – in the same location as the path to the Android SDK tools, untick 'Gradle Installed with Unity' (Figure 2.2) and set the Gradle path to the location of the downloaded version. For the Gradle build to work correctly, the Gradle version must be below Gradle 7 as anything above is not compatible with the current build settings of the app.

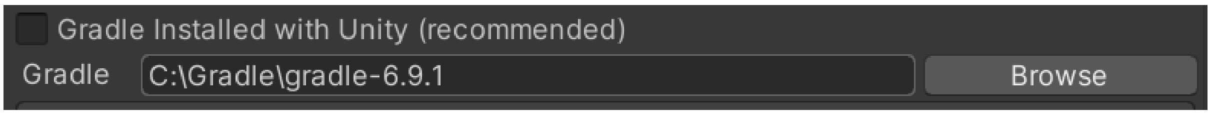

Gradle Installed with Unity (recommended)
Gradle C:\Gradle\gradle-6.9.1

Figure 2.2: Gradle path location

2.2 App Installation

The app is available in two different forms. The app can either be downloaded as an .apk and used on an Android phone (described in Section 2.2.1) or for app development, the Unity package can be downloaded and imported into Unity to access the source code and make future changes (described in Section 2.2.2).

2.2.1 Installation for Users

The app can be downloaded from *Dropbox* using the following link. Click on 'or continue to website' and then on 'Download' and the download will start automatically after confirming that the file is trusted. After downloading the app, click on *install* and the installation process should start. After successful download and installation, the app can then be opened, and the user can start by creating an account, or if an account already exists, login. The screenshots in Figures 2.3 and 2.4 show the installation process:

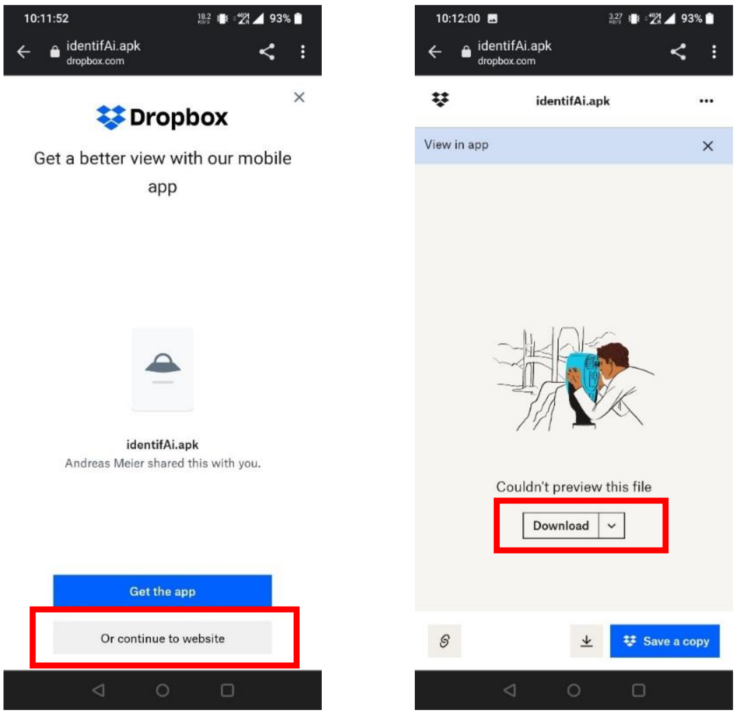


Figure 2.3 & 2.4: Dropbox download instructions

2.2.2 Installation for Development

The app is available as a Unity package following the following link. All software requirements mentioned in section 2.1 should be met. If that is the case, the package can be imported into Unity. After importing the package, the entire source code is made available, and changes can be made to the app. Figures 2.5 – 2.8 show step by step how to import a package into Unity.

1. Open *Unity Hub* and create a new project by clicking 'New project'. After that, select the empty 3D template and open the new project. (Figure 2.5)

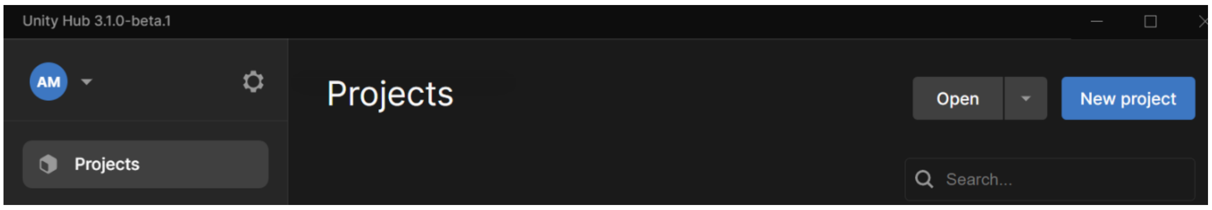


Figure 2.5: Opening a new project in Unity Hub

2. Then, click on "Assets", "Import Package" and "Custom Package". (Figure 2.6)

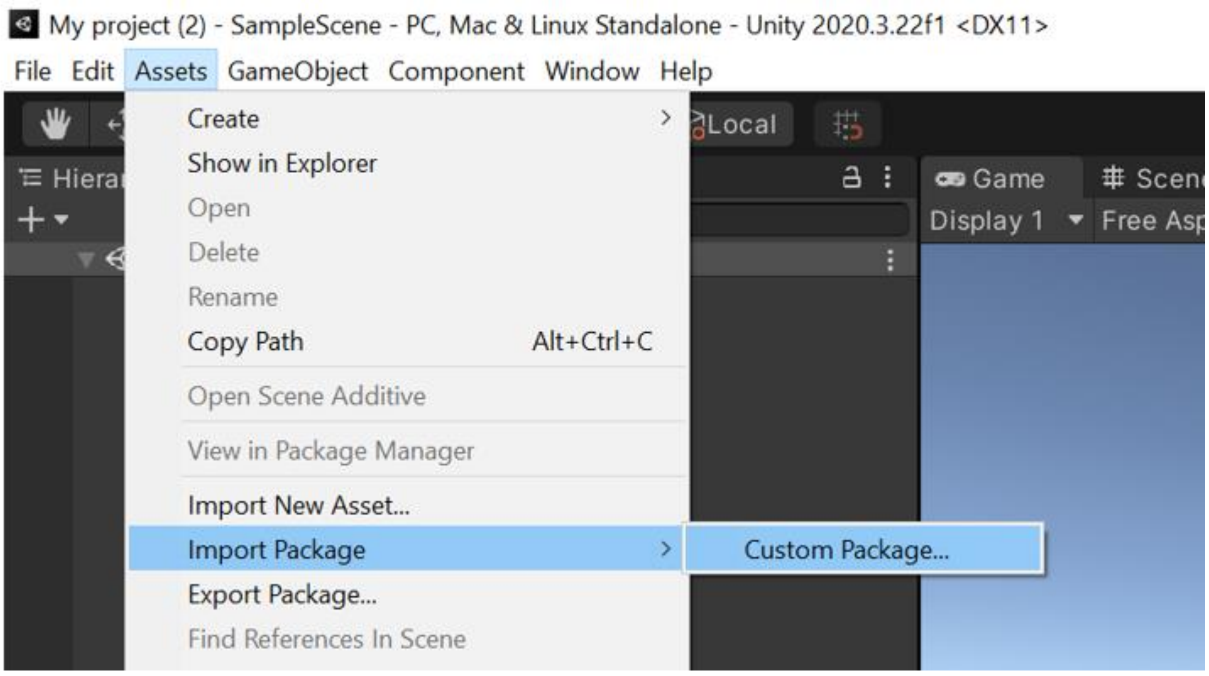
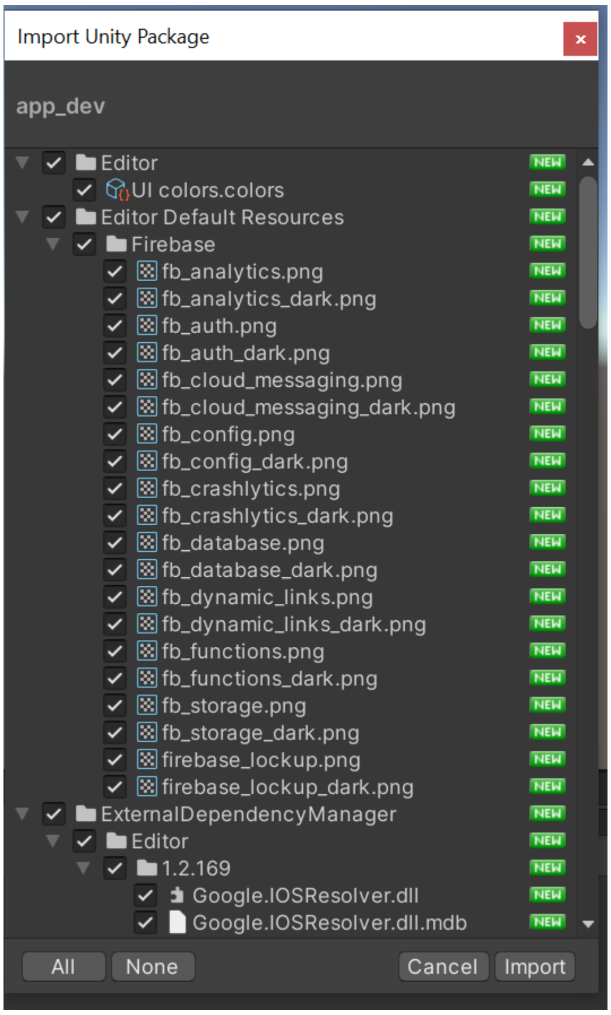


Figure 2.6: Importing a custom package in Unity

3. Select the Unity package file that you have downloaded from Dropbox and open it. (Figure 2.7)

Figure 2.7: Downloaded unity package file from Dropbox

4. The window in Figure 2.8 will then be opened automatically. Make sure every folder is ticked and click ‘Import’.



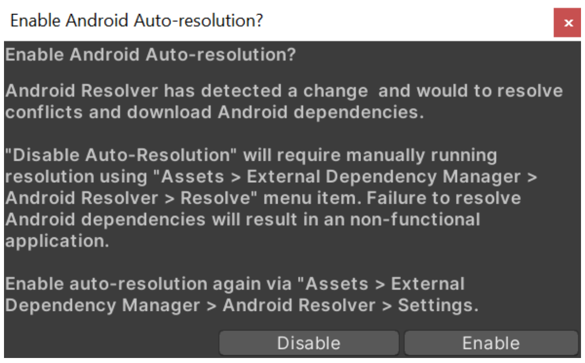
Import Unity Package

app_dev

- ✓ Editor NEW
  - ✓ UI colors.colors NEW
- ✓ Editor Default Resources NEW
  - ✓ Firebase NEW
    - ✓ fb_analytics.png NEW
    - ✓ fb_analytics_dark.png NEW
    - ✓ fb_auth.png NEW
    - ✓ fb_auth_dark.png NEW
    - ✓ fb_cloud_messaging.png NEW
    - ✓ fb_cloud_messaging_dark.png NEW
    - ✓ fb_config.png NEW
    - ✓ fb_config_dark.png NEW
    - ✓ fb_crashlytics.png NEW
    - ✓ fb_crashlytics_dark.png NEW
    - ✓ fb_database.png NEW
    - ✓ fb_database_dark.png NEW
    - ✓ fb_dynamic_links.png NEW
    - ✓ fb_dynamic_links_dark.png NEW
    - ✓ fb_functions.png NEW
    - ✓ fb_functions_dark.png NEW
    - ✓ fb_storage.png NEW
    - ✓ fb_storage_dark.png NEW
    - ✓ firebase_lockup.png NEW
    - ✓ firebase_lockup_dark.png NEW
- ✓ ExternalDependencyManager NEW
  - ✓ Editor NEW
    - ✓ 1.2.169 NEW
      - ✓ Google.IOSResolver.dll NEW
      - ✓ Google.IOSResolver.dll.mdb NEW

Figure 2.8: Importing all packages from the downloaded file into Unity

5. Ensure the platform is set to Android under Build Settings. If it is not selected as default, select Android and ‘switch platform’.
   a. If the pop-up window in Figure 2.9 appears, after switching platform, choose to ‘Enable’ Android Auto-resolution.


Enable Android Auto-resolution?

Enable Android Auto-resolution?

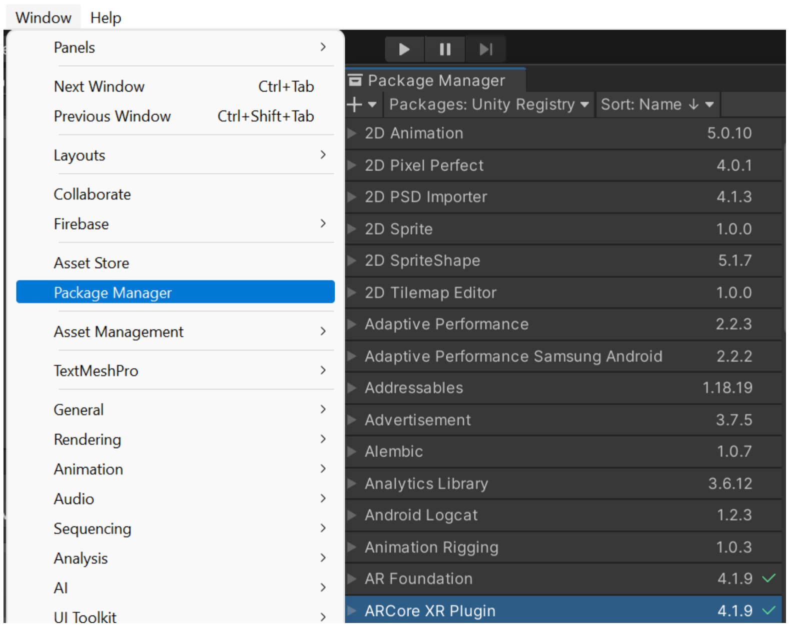
Android Resolver has detected a change and would to resolve conflicts and download Android dependencies.

"Disable Auto-Resolution" will require manually running resolution using "Assets > External Dependency Manager > Android Resolver > Resolve" menu item. Failure to resolve Android dependencies will result in an non-functional application.

Enable auto-resolution again via "Assets > External Dependency Manager > Android Resolver > Settings.

Figure 2.9: Pop-up Window to enable Android Auto-Resolution

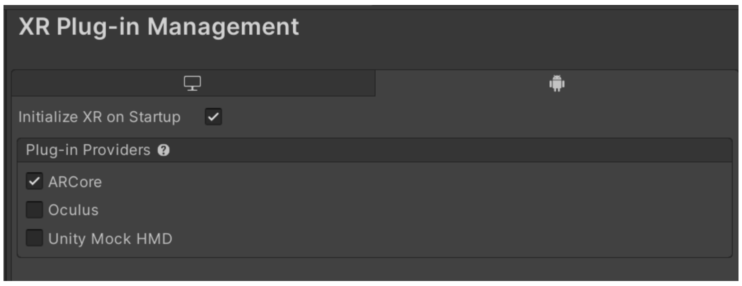
6. Under the ‘Window’ tab, select Package Manager. Within the dropdown menu under the new Package Manager window, select ‘Unity Registry’, displayed in Figure 2.10. Here, find and install both the *AR Foundation* and *ARCore XR Plugin* packages.



2D Animation	5.0.10
2D Pixel Perfect	4.0.1
2D PSD Importer	4.1.3
2D Sprite	1.0.0
2D SpriteShape	5.1.7
2D Tilemap Editor	1.0.0
Adaptive Performance	2.2.3
Adaptive Performance Samsung Android	2.2.2
Addressables	1.18.19
Advertisement	3.7.5
Alembic	1.0.7
Analytics Library	3.6.12
Android Logcat	1.2.3
Animation Rigging	1.0.3
AR Foundation	4.1.9 ✓
ARCore XR Plugin	4.1.9 ✓

Figure 2.10: Installing AR packages

7. Under Project Settings, open XR Plug-in Management side tab as shown in Figure 2.11. Set the mobile development kit to Android and select the Plug-in Providers to ARCore.


XR Plug-in Management

Initialize XR on Startup ✓

Plug-in Providers

✓ ARCore
☐ Oculus
☐ Unity Mock HMD

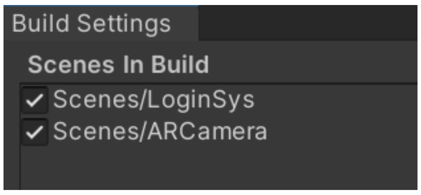
Figure 2.11: XR Plug-in Management tab

8. After importing all the packages, contact a developer and access to the Firestore database can be granted. The contact details can be found in Section 4.1.3. Alternatively, the provided Firestore account and Unity Hub account (also found in Section 4.1.3) can be used to open the current project for easier access.

2.2.3 Building to Android

For developers wanting to make additions to the application and create an upgraded version, the following section instructs how to build and run it from Unity. An Android phone is required in the building process.

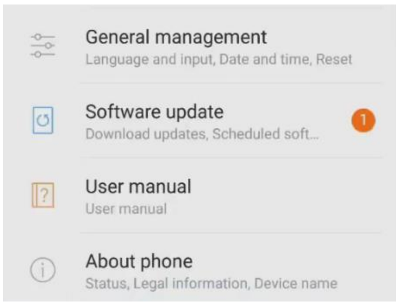
1. In Unity, add all current and new open scenes to 'Scenes in Build', as shown in Figure 2.12.



Build Settings
Scenes In Build
✓ Scenes/LoginSys
✓ Scenes/ARCamera

Figure 2.12: Open scenes to be built

2. On the Android phone's settings scroll down to and select 'About phone' (Figure 2.13).



General management
Language and input, Date and time, Reset

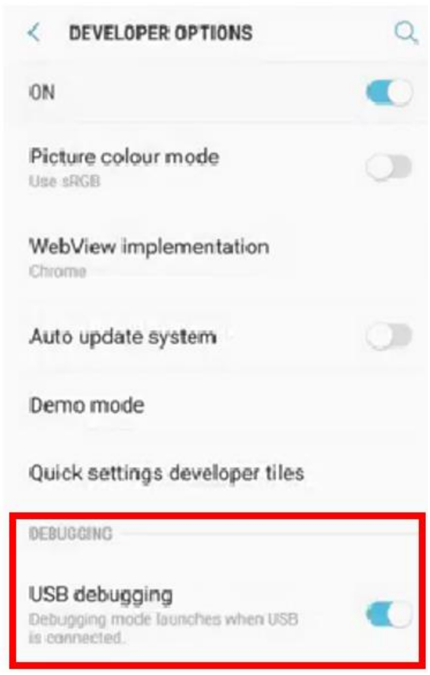
Software update
Download updates, Scheduled soft...

User manual
User manual

About phone
Status, Legal information, Device name

Figure 2.13: Android settings

3. Locate the mobile's 'Build Number' (location varies between devices). Tap the build number 7 times, which will prompt you with a PIN input. After entering your PIN this will enable developer mode.
4. In developer options, enable USB debugging (Figure 2.14).


DEVELOPER OPTIONS

ON

Picture colour mode
Use sRGB

WebView implementation
Chrome

Auto update system

Demo mode

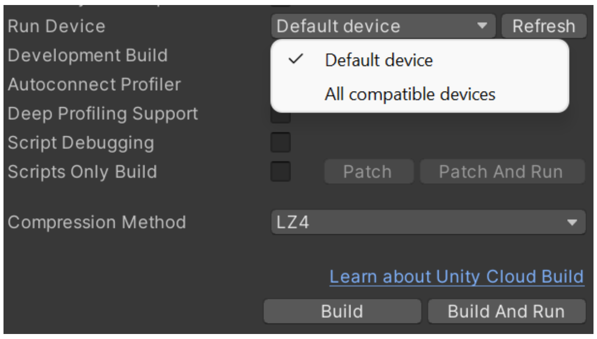
Quick settings developer tiles

DEBUGGING

USB debugging
Debugging mode launches when USB is connected.

Figure 2.14: USB debugging option in developer options

5. Connect the mobile to your PC running Unity via USB. Under ‘Build Settings’, click ‘refresh’ and your device type should be displayed under the default device dropdown (Figure 2.15).


Run Device: Default device — Refresh
- ✓ Default device
- All compatible devices

Development Build
Autoconnect Profiler
Deep Profiling Support
Script Debugging
Scripts Only Build — Patch — Patch And Run

Compression Method: LZ4

Learn about Unity Cloud Build

Build — Build And Run

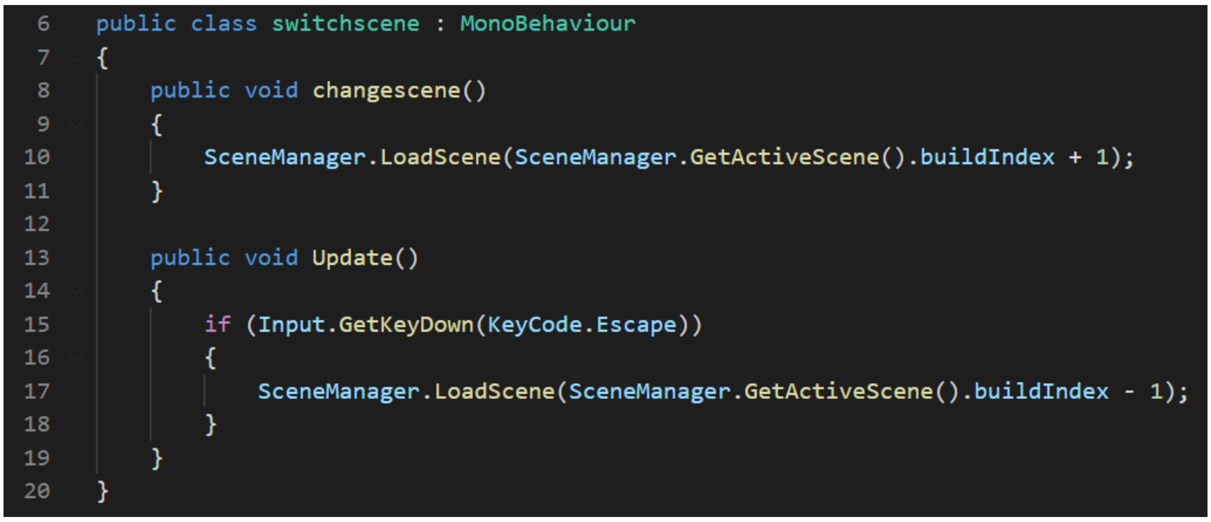
Figure 2.15: Device options under build settings

6. Select ‘Build and Run’. Here, a new window will appear in which you will need to name and save the new APK file which will then automatically build to the device and run, with the device still connected.

3 User Operations and Backend Code for UI

3.1 App components and navigation

The user interface is created in Unity, containing two scenes: the Login scene and the AR scene. The Login scene consists of all pages for logging in and creating an account, as well as managing the user's profile page once logged in. The AR scene opens the device's camera and initialises the AR environment. The buttons created from the frontend in Unity are connected to C# scripts, containing the code for the functions of each button. The C# scripts provide the logic needed for a functioning UI, which are explained in depth in this section. To navigate between the two scenes, switchscene.cs has been written, shown in Figure 3.1.



```
public class switchscene : MonoBehaviour
{
    public void changescene()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
    }

    public void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
          SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex - 1);
        }
    }
}
```

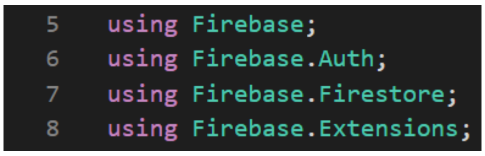
Figure 3.1: changescene and Update function

The *changescene* function is linked to the AR button on the profile page. The Login scene is set as Scene 0 (the default scene) and the AR scene is set as Scene 1. This function increases the scene by 1, changing the scene that is currently displayed on the device from Scene 0 to Scene 1. The *Update* function is always running while the app is running: this function allows the back button on Android phones to return the scene to the profile page by decreasing the scene from 1 to 0.

3.2 Firebase and C# Script Setup

To link the Firebase functionalities to the C# scripts, the following code in Figures 3.2 – 3.4 is required:

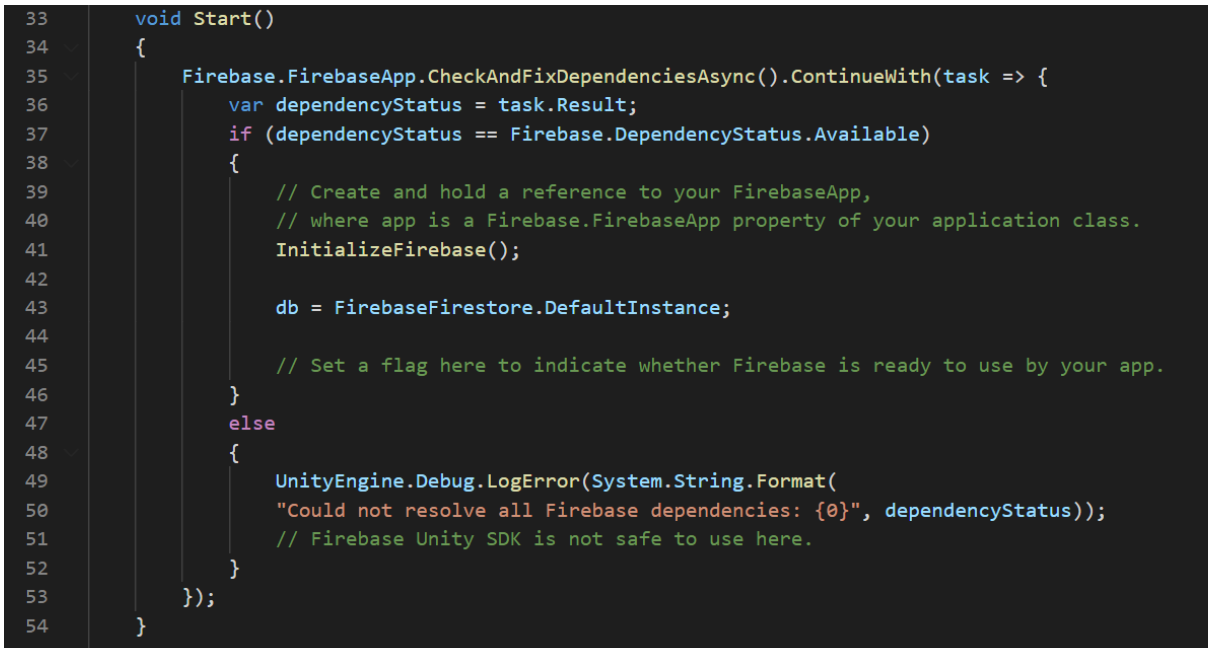
At the top of the C# script, the relevant products should be initialised (for Firebase Authentication and Firebase Firestore) (Figure 3.2).



```csharp
using Firebase;
using Firebase.Auth;
using Firebase.Firestore;
using Firebase.Extensions;
```

Figure 3.2: Initialization of Firebase products

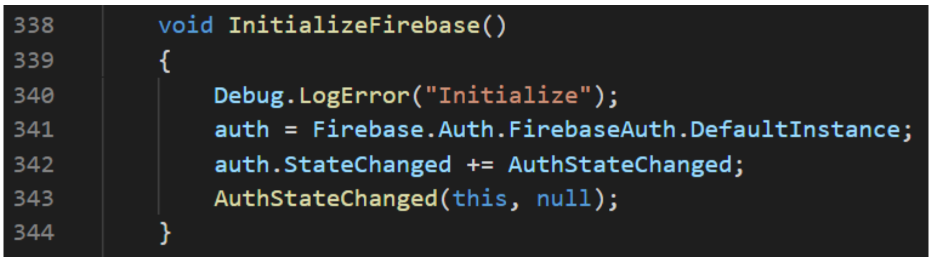
The following code in Figure 3.3 is also included to ensure Google Play Services is up to date, since this is required for Firebase Unity SDK:



```csharp
void Start()
{
    Firebase.FirebaseApp.CheckAndFixDependenciesAsync().ContinueWith(task => {
        var dependencyStatus = task.Result;
        if (dependencyStatus == Firebase.DependencyStatus.Available)
        {
            // Create and hold a reference to your FirebaseApp,
            // where app is a Firebase.FirebaseApp property of your application class.
            InitializeFirebase();

            db = FirebaseFirestore.DefaultInstance;

            // Set a flag here to indicate whether Firebase is ready to use by your app.
        }
        else
        {
            UnityEngine.Debug.LogError(System.String.Format(
            "Could not resolve all Firebase dependencies: {0}", dependencyStatus));
            // Firebase Unity SDK is not safe to use here.
        }
    });
}
```

Figure 3.3: Start function

The *InitializeFirebase* function is called in Figure 3.3, which holds a reference to the app and ensures Firebase is initialised (Figure 3.4).



```csharp
void InitializeFirebase()
{
    Debug.LogError("Initialize");
    auth = Firebase.Auth.FirebaseAuth.DefaultInstance;
    auth.StateChanged += AuthStateChanged;
    AuthStateChanged(this, null);
}
```

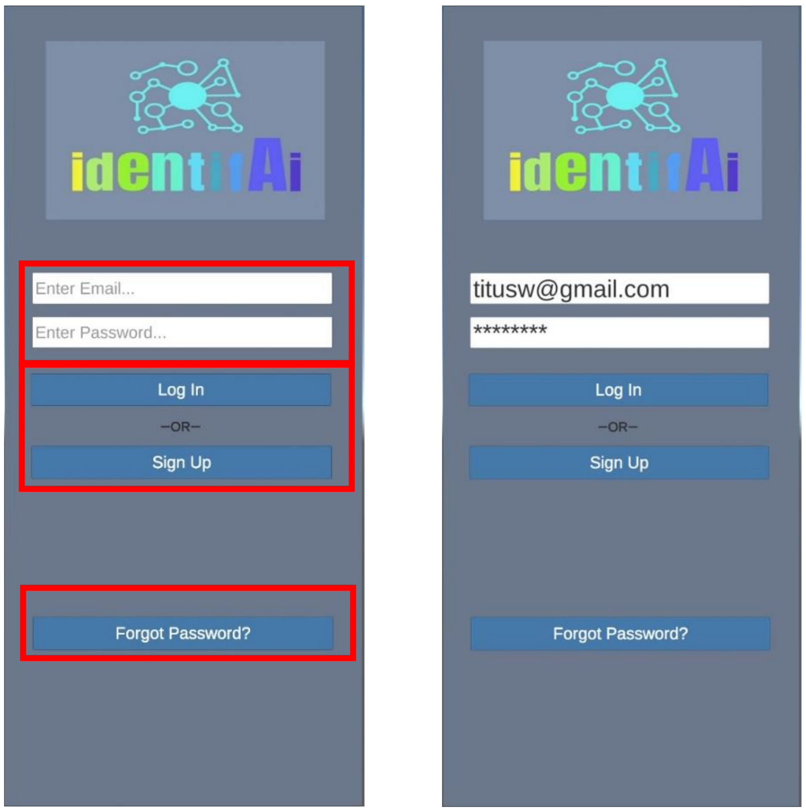
Figure 3.4: InitializeFirebase function

3.3 Login Scene

The C# code for the login scene is saved in *Assets/Scripts/FirebaseController.cs*. The screenshots on the left of the page show the different pages of the app without any input, while the right shows an example input for each input field.

3.3.1 Login Page

The login page is the default page displayed when the app is first opened, shown in Figures 3.5 and 3.6. If a user has already created an account, the user can then proceed to login with the according email and password. After entering the user's details, press the "Log In" button. To create an account, press the 'Sign Up' button, opening the Sign Up page. If the user has forgotten the password to log in to their account, the password can be reset by clicking on 'Forgot Password?', which opens the Forgotten Password Page.

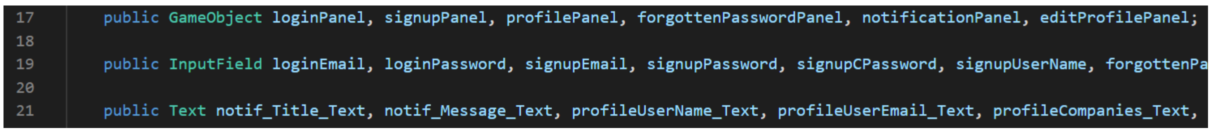


titusw@gmail.com

****

Figure 3.5 & 3.6: Login interface without and with example input

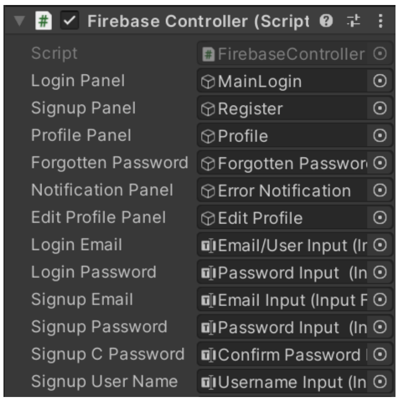
In the beginning of the script (Figure 3.7), the panels for each page are set as GameObjects (line 17), as well as names for each input field that can be accessed to give each input different rules (line 19). There are also placeholders for any text that is updated depending on the user, saved as Text objects (line 21).



```
    public GameObject loginPanel, signupPanel, profilePanel, forgottenPasswordPanel, notificationPanel, editProfilePanel;

    public InputField loginEmail, loginPassword, signupEmail, signupPassword, signupCPassword, signupUserName, forgottenPa

    public Text notif_Title_Text, notif_Message_Text, profileUserName_Text, profileUserEmail_Text, profileCompanies_Text,
```

Figure 3.7: Object and input field creation

The pages, input boxes and placeholders are linked to the relative components in Unity: under the FirebaseController object in the Login scene, each component has been dragged to their corresponding variable in the C# script, shown in Figure 3.8.



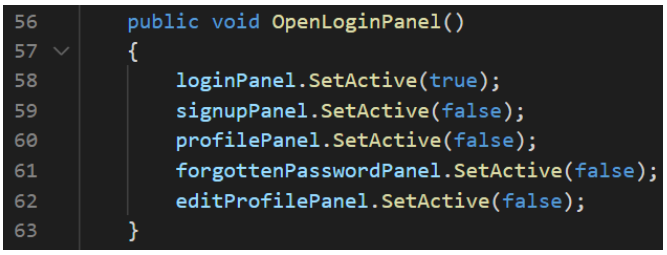
Firebase Controller (Script

Script	FirebaseController
Login Panel	MainLogin
Signup Panel	Register
Profile Panel	Profile
Forgotten Password	Forgotten Passwor
Notification Panel	Error Notification
Edit Profile Panel	Edit Profile
Login Email	Email/User Input (Ir
Login Password	Password Input (In
Signup Email	Email Input (Input F
Signup Password	Password Input (In
Signup C Password	Confirm Password
Signup User Name	Username Input (In

Figure 3.8: Connecting C# scripts with UI

There are also functions to open each panel (Figure 3.9), where *SetActive* is set to true for the corresponding panel. For example, for *OpenLoginPanel*, *loginPanel* is set to active and all other panels are set to false. This means that that the panel with *SetActive* set to true is displayed on the device screen, whilst the others set to false are hidden.

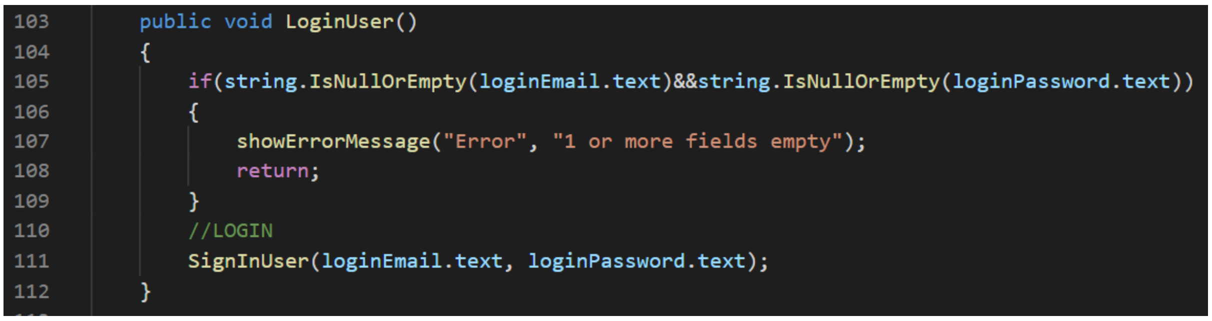
Within the user interface, these functions are linked to the relevant buttons which execute the function to switch pages when clicked, for example from the ‘Forgotten Password’ page back to the login screen. This is linked in Unity within the same section as shown previously (Figure 3.8).



```
    public void OpenLoginPanel()
    {
        loginPanel.SetActive(true);
        signupPanel.SetActive(false);
        profilePanel.SetActive(false);
        forgottenPasswordPanel.SetActive(false);
        editProfilePanel.SetActive(false);
    }
```

Figure 3.9: Activity status for OpenLoginPanel

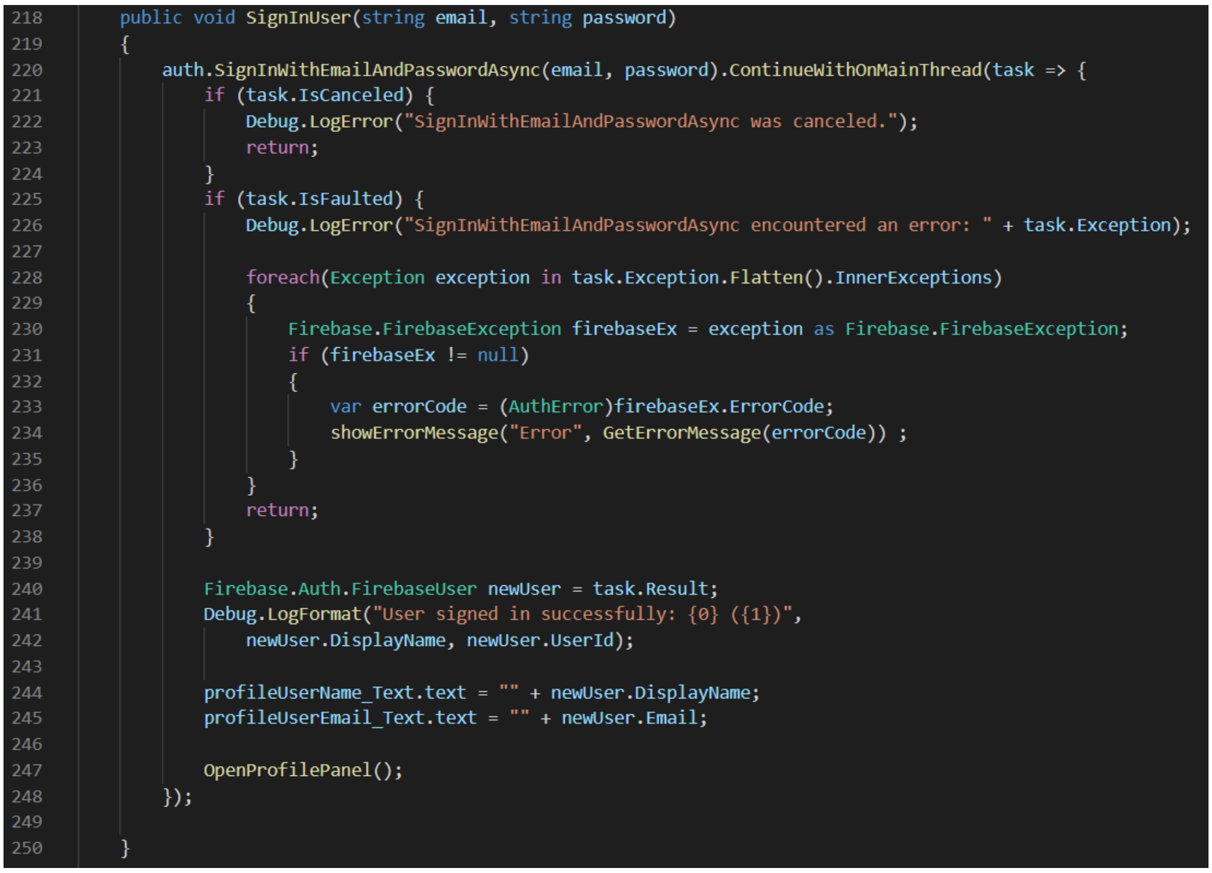
The following functions (Figures 3.10 – 3.11) are designed for the login process on the Login page. The *LoginUser* function (Figure 3.10) is executed when the Login button is pressed. In this function, rules are set to handle empty inputs and give the corresponding error message, then exit the function if this error occurs. Otherwise, another function, *SignInUser* (Figure 3.11), is called which carries out the login process with the input given by the user.



```
public void LoginUser()
{
    if(string.IsNullOrEmpty(loginEmail.text)&&string.IsNullOrEmpty(loginPassword.text))
    {
        showErrorMessage("Error", "1 or more fields empty");
        return;
    }
    //LOGIN
    SignInUser(loginEmail.text, loginPassword.text);
}
```

Figure 3.10: LoginUser function

Firebase uses this *SignInUser* function to login users. The *SignInWithEmailAndPasswordAsync* function is provided through Firebase Authentication: it takes the provided email and password and checks if the login credentials are correct (line 228), and if not handles the exception accordingly (lines 229-245) by displaying the error in the console and providing an error notification to the user (line 242). If the login is successful, the current user is set for Firebase Authentication (line 248) and the placeholders for the user’s name and email in the profile page is set (lines 252-253). Finally, the function opens the personal profile page (line 256).



```
218    public void SignInUser(string email, string password)
219    {
220        auth.SignInWithEmailAndPasswordAsync(email, password).ContinueWithOnMainThread(task => {
221            if (task.IsCanceled) {
222                Debug.LogError("SignInWithEmailAndPasswordAsync was canceled.");
223                return;
224            }
225            if (task.IsFaulted) {
226                Debug.LogError("SignInWithEmailAndPasswordAsync encountered an error: " + task.Exception);
227
228                foreach(Exception exception in task.Exception.Flatten().InnerExceptions)
229                {
230                    Firebase.FirebaseException firebaseEx = exception as Firebase.FirebaseException;
231                    if (firebaseEx != null)
232                    {
233                        var errorCode = (AuthError)firebaseEx.ErrorCode;
234                        showErrorMessage("Error", GetErrorMessage(errorCode)) ;
235                    }
236                }
237                return;
238            }
239
240            Firebase.Auth.FirebaseUser newUser = task.Result;
241            Debug.LogFormat("User signed in successfully: {0} ({1})",
242                newUser.DisplayName, newUser.UserId);
243
244            profileUserName_Text.text = "" + newUser.DisplayName;
245            profileUserEmail_Text.text = "" + newUser.Email;
246
247            OpenProfilePanel();
248        });
249
250    }
```

Figure 3.11: SignInUser function

3.3.2 Sign Up Page

The Sign Up Page can be accessed from the Login page. To create an account, the user enters their username, email, password and confirms their password in the last input field. The 'Sign Up' button is then clicked, and an account will be created, allowing the user to log into identifAi. To navigate back to the login menu, the "Back" button is pressed. This screen is shown in Figures 3.12 and 3.13.

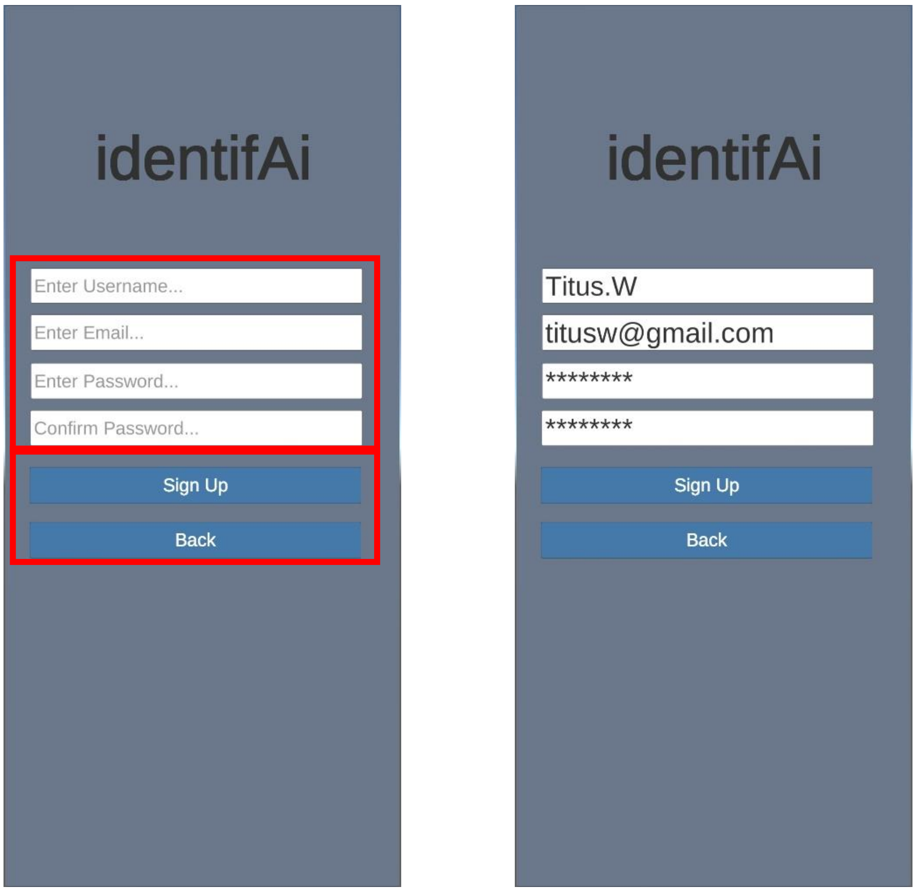
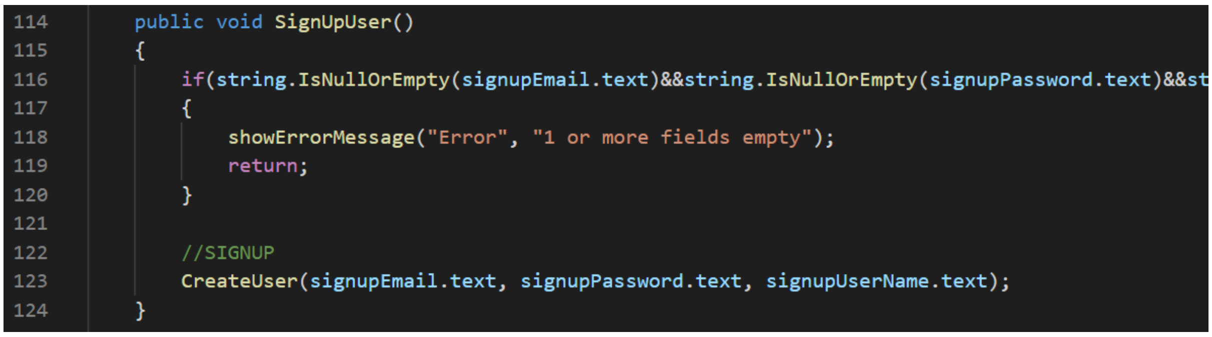


Figure 3.12 & 3.13: Sign Up interface without and with example input

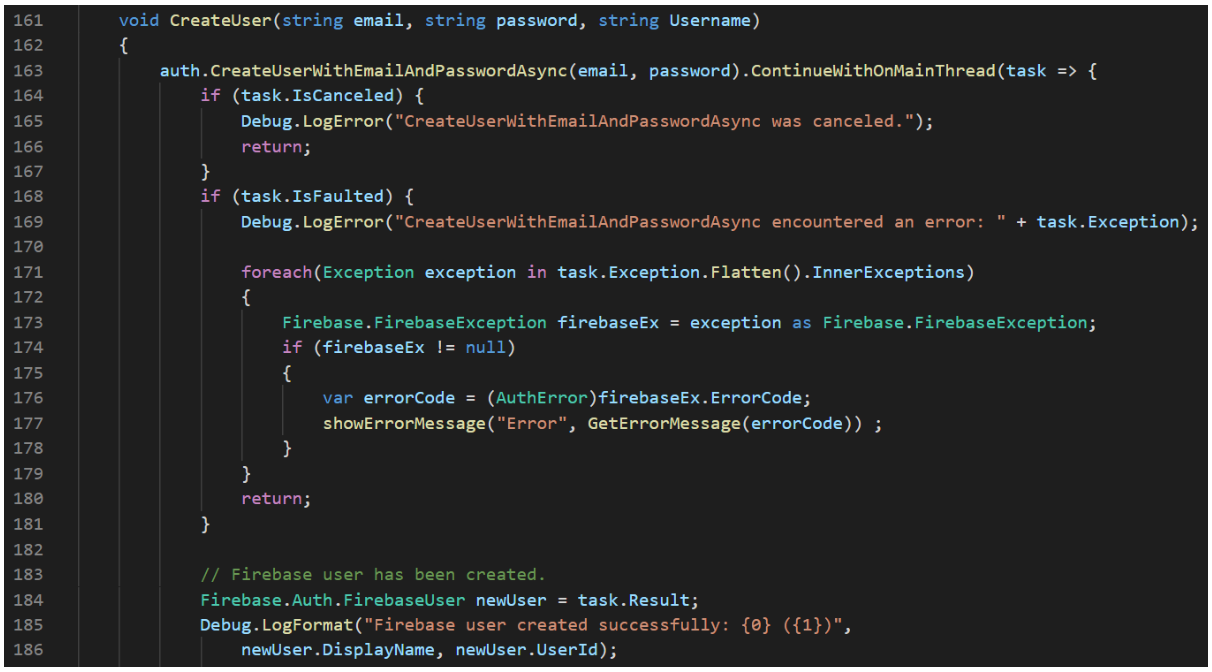
The function *SignUpUser* (Figure 3.14) is executed when the 'Sign Up' button is pressed. It checks if the different input fields are empty (line 114), and if true displays the according error message. Within this function, another function, *CreateUser* is called, which creates a new account if no empty inputs were detected by *SignUpUser*.



```
public void SignUpUser()
{
    if(string.IsNullOrEmpty(signupEmail.text)&&string.IsNullOrEmpty(signupPassword.text)&&st
    {
        showErrorMessage("Error", "1 or more fields empty");
        return;
    }

    //SIGNUP
    CreateUser(signupEmail.text, signupPassword.text, signupUserName.text);
}
```

Figure 3.14: SignUpUser function

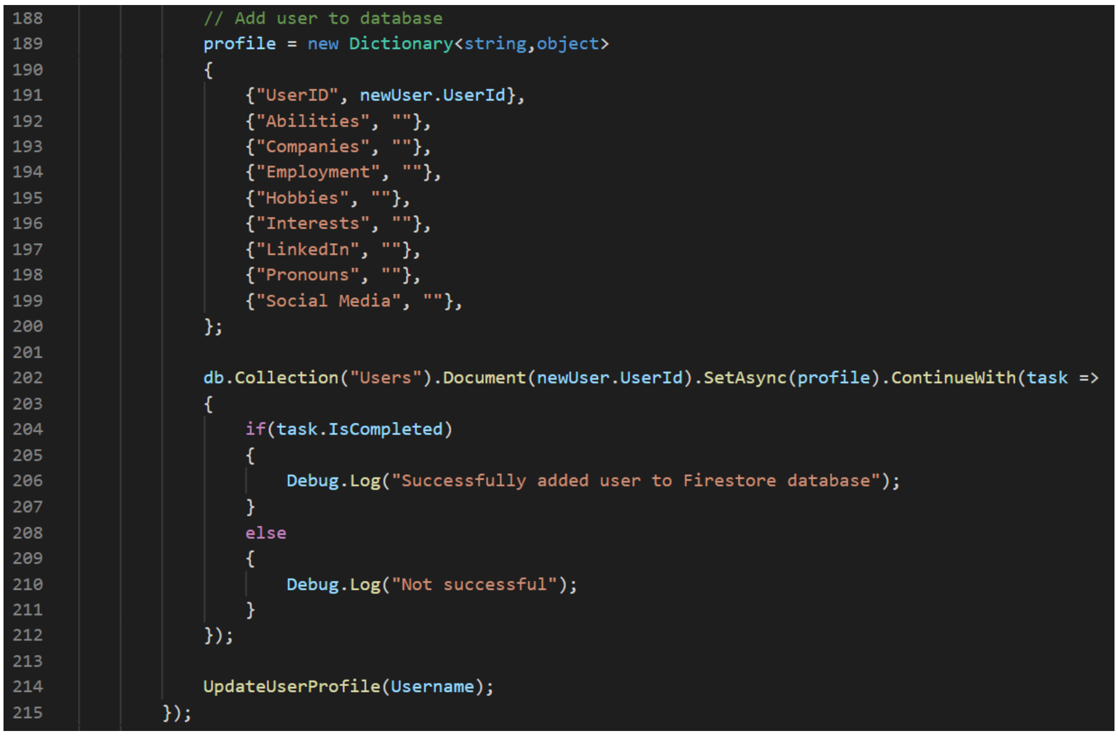
CreateUser (Figure 3.15) is used by Firebase to create new accounts, and takes in the email, password, and username as inputs. The *CreateUserWithEmailAndPasswordAsync* function is provided by Firebase Authentication to create the account if all requirements are met: the function checks for errors, for example if the email is already used by another user and handles the exception accordingly with different error messages. If no exceptions are found, Firebase Authentication creates the account.



```
void CreateUser(string email, string password, string Username)
{
    auth.CreateUserWithEmailAndPasswordAsync(email, password).ContinueWithOnMainThread(task => {
        if (task.IsCanceled) {
            Debug.LogError("CreateUserWithEmailAndPasswordAsync was canceled.");
            return;
        }
        if (task.IsFaulted) {
            Debug.LogError("CreateUserWithEmailAndPasswordAsync encountered an error: " + task.Exception);

            foreach(Exception exception in task.Exception.Flatten().InnerExceptions)
            {
                Firebase.FirebaseException firebaseEx = exception as Firebase.FirebaseException;
                if (firebaseEx != null)
                {
                    var errorCode = (AuthError)firebaseEx.ErrorCode;
                    showErrorMessage("Error", GetErrorMessage(errorCode)) ;
                }
            }
            return;
        }

        // Firebase user has been created.
        Firebase.Auth.FirebaseUser newUser = task.Result;
        Debug.LogFormat("Firebase user created successfully: {0} ({1})",
            newUser.DisplayName, newUser.UserId);
```

Figure 3.15: CreateUser function

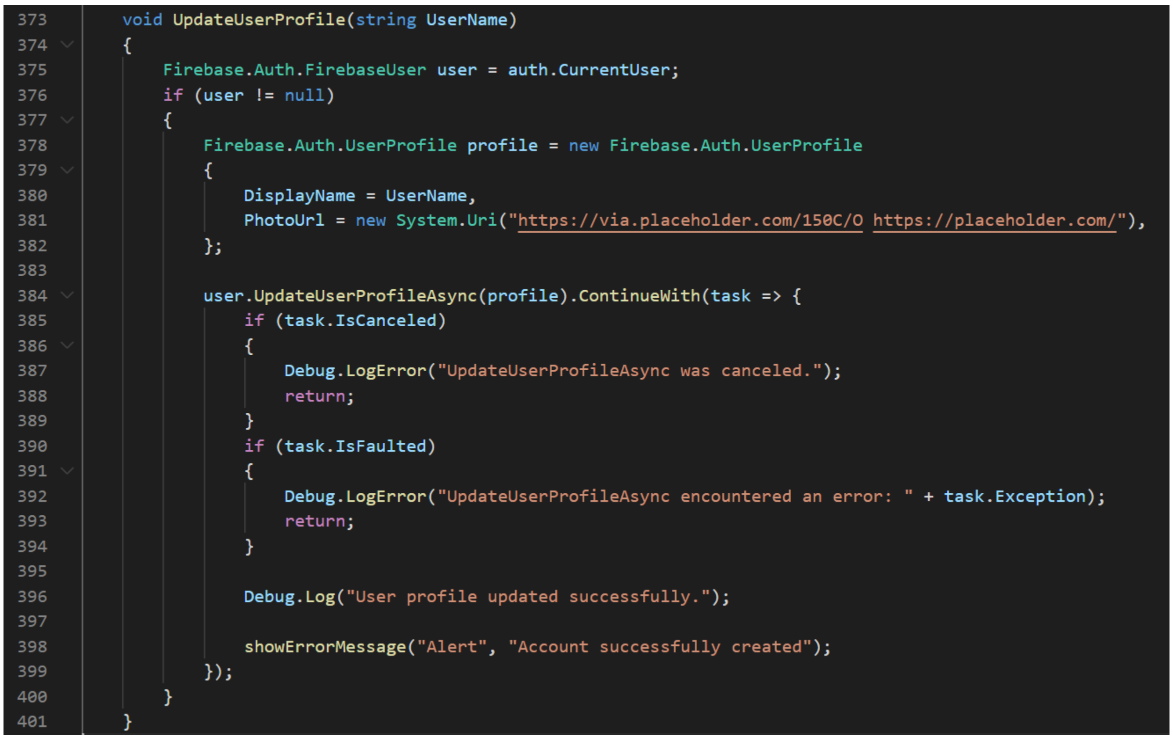
If the account is successfully created by Firebase Authentication, the account is also set up in the Firebase Firestore cloud database (Figure 3.16), for the user's information to be edited. A new dictionary object is created to be added to the user's profile in the database (lines 189 – 200). The fields for the personal profile are initialised as an empty string to be edited later when the user edits their profile. The user's empty profile is added to the database under their user ID in lines 202 – 212 and finally the *UpdateUserProfile* function is called.



```
                // Add user to database
                profile = new Dictionary<string,object>
                {
                    {"UserID", newUser.UserId},
                    {"Abilities", ""},
                    {"Companies", ""},
                    {"Employment", ""},
                    {"Hobbies", ""},
                    {"Interests", ""},
                    {"LinkedIn", ""},
                    {"Pronouns", ""},
                    {"Social Media", ""},
                };

                db.Collection("Users").Document(newUser.UserId).SetAsync(profile).ContinueWith(task =>
                {
                    if(task.IsCompleted)
                    {
                        Debug.Log("Successfully added user to Firestore database");
                    }
                    else
                    {
                        Debug.Log("Not successful");
                    }
                });

                UpdateUserProfile(Username);
            });
```

Figure 3.16: Dictionary to save personal profile data

The *UpdateUserProfile* function (Figure 3.17) handles any errors if the app is disconnected and ensures that the correct user is set as the current user in Firestore. The function checks if the user is not set to null (so the user exists) and if so, updates the user profile within Firebase Authentication.

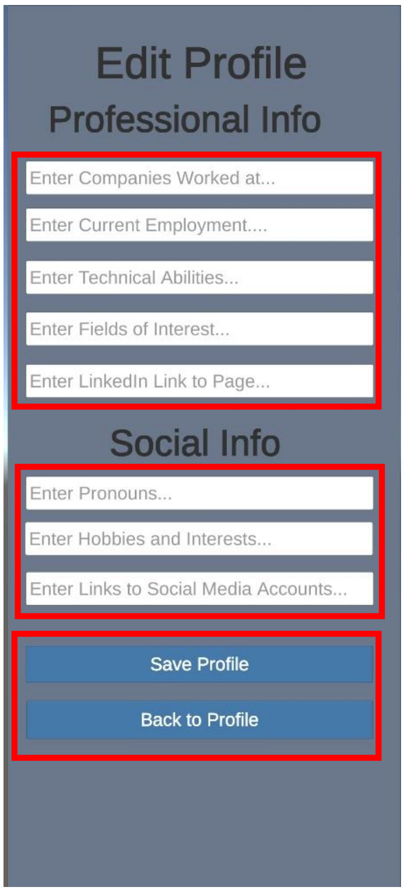


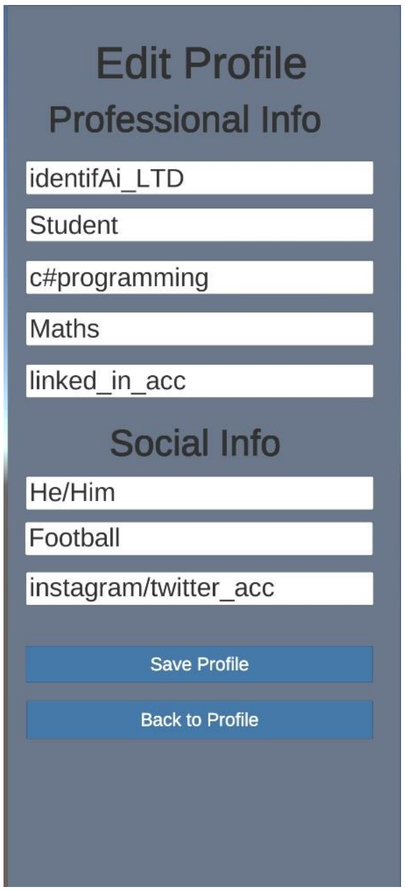
```
    void UpdateUserProfile(string UserName)
    {
        Firebase.Auth.FirebaseUser user = auth.CurrentUser;
        if (user != null)
        {
            Firebase.Auth.UserProfile profile = new Firebase.Auth.UserProfile
            {
                DisplayName = UserName,
                PhotoUrl = new System.Uri("https://via.placeholder.com/150C/O https://placeholder.com/"),
            };

            user.UpdateUserProfileAsync(profile).ContinueWith(task => {
                if (task.IsCanceled)
                {
                    Debug.LogError("UpdateUserProfileAsync was canceled.");
                    return;
                }
                if (task.IsFaulted)
                {
                    Debug.LogError("UpdateUserProfileAsync encountered an error: " + task.Exception);
                    return;
                }

                Debug.Log("User profile updated successfully.");

                showErrorMessage("Alert", "Account successfully created");
            });
        }
    }
```

Figure 3.17: UpdateUserProfile function

3.3.3 Profile Page / Edit Profile Page

On a user's personal profile page, their personal and social information is displayed, for example in Figure 3.19. The professional information a user is able to add include companies worked for, current employment, technical abilities, fields of interest and the name of a *LinkedIn* page. For social information, a user can add their pronouns, hobbies and interests, and names of other social media accounts. This information can be edited by pressing the 'Edit' button, which opens the Edit Profile page, shown in Figure 3.18. This page contains input boxes for each field, which is saved to the user's profile when the 'Save Profile' button is pressed. The "Back to Profile" button returns to the Profile page and the changes to the personal profile should be displayed, shown in Figure 3.20.


Edit Profile
Professional Info
Enter Companies Worked at...
Enter Current Employment....
Enter Technical Abilities...
Enter Fields of Interest...
Enter LinkedIn Link to Page...
Social Info
Enter Pronouns...
Enter Hobbies and Interests...
Enter Links to Social Media Accounts...
Save Profile
Back to Profile


Edit Profile
Professional Info
identifAi_LTD
Student
c#programming
Maths
linked_in_acc
Social Info
He/Him
Football
instagram/twitter_acc
Save Profile
Back to Profile

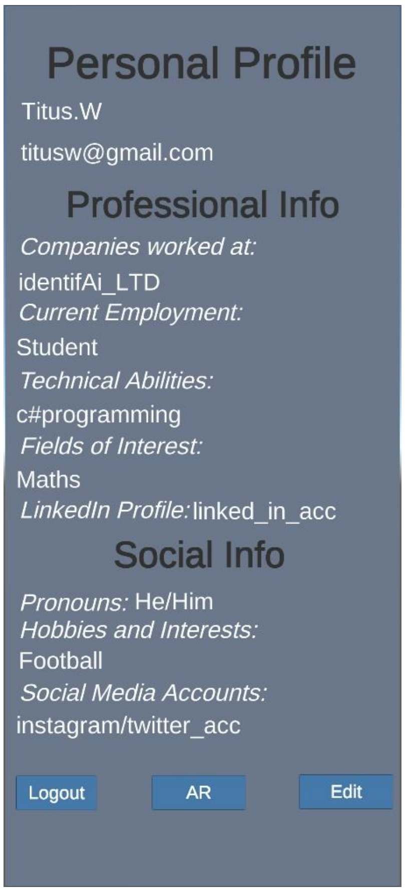
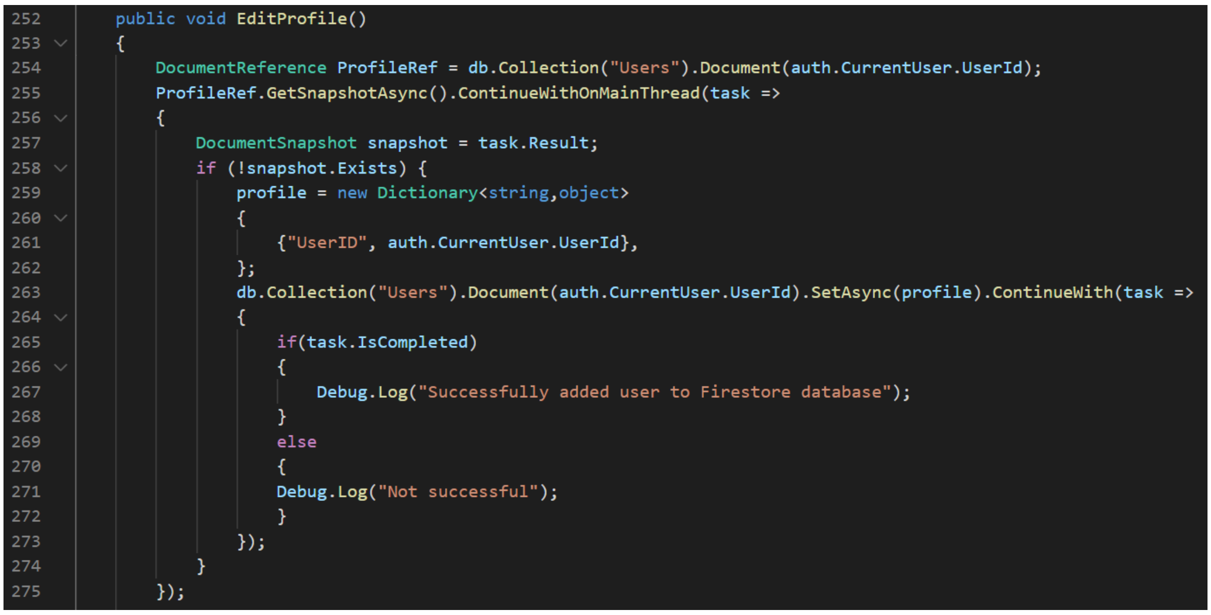

Personal Profile
Titus.W
titusw@gmail.com
Professional Info
Companies worked at:
identifAi_LTD
Current Employment:
Student
Technical Abilities:
c#programming
Fields of Interest:
Maths
LinkedIn Profile: linked_in_acc
Social Info
Pronouns: He/Him
Hobbies and Interests:
Football
Social Media Accounts:
instagram/twitter_acc

Figure 3.18, 3.19 & 3.20: Edit profile interface without and with example input. Personal profile after updating information with example input.

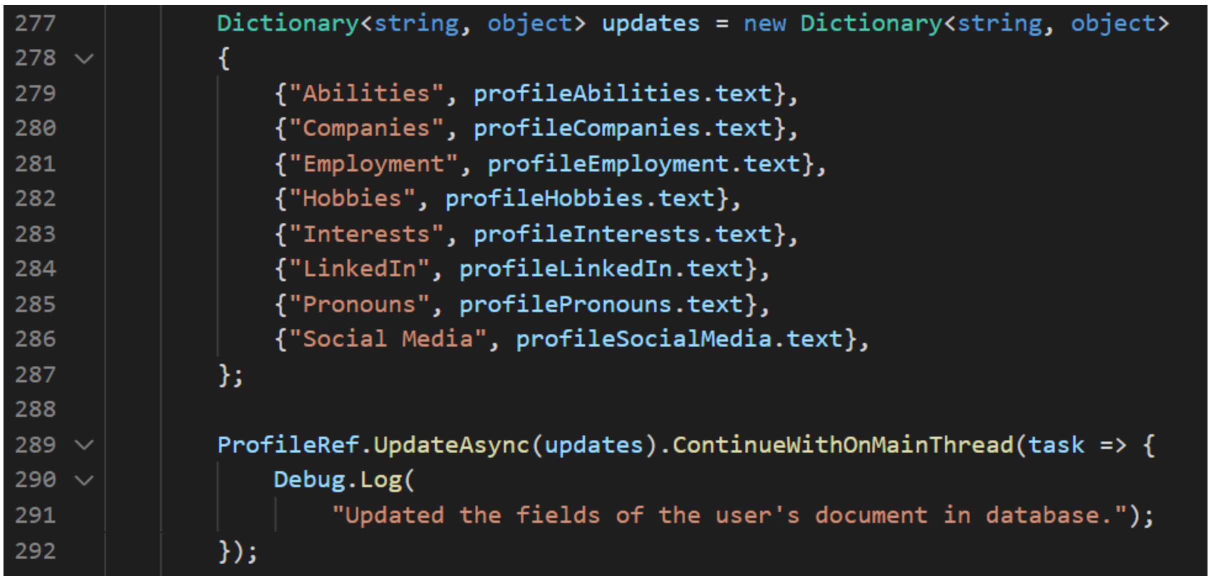
The function *EditProfile* (Figure 3.21) is written to handle updating the information in the Firestore database concerning the user's profile information and is linked to the 'Save Profile' button. This function first checks that the user's profile exists in Firestore by checking their user ID: if their profile does not exist in the database for some reason, their account is added as a document in the database under their user ID.



```
public void EditProfile()
{
    DocumentReference ProfileRef = db.Collection("Users").Document(auth.CurrentUser.UserId);
    ProfileRef.GetSnapshotAsync().ContinueWithOnMainThread(task =>
    {
        DocumentSnapshot snapshot = task.Result;
        if (!snapshot.Exists) {
            profile = new Dictionary<string,object>
            {
                {"UserID", auth.CurrentUser.UserId},
            };
            db.Collection("Users").Document(auth.CurrentUser.UserId).SetAsync(profile).ContinueWith(task =>
            {
                if(task.IsCompleted)
                {
                    Debug.Log("Successfully added user to Firestore database");
                }
                else
                {
                Debug.Log("Not successful");
                }
            });
        }
    });
```

Figure 3.21: EditProfile function

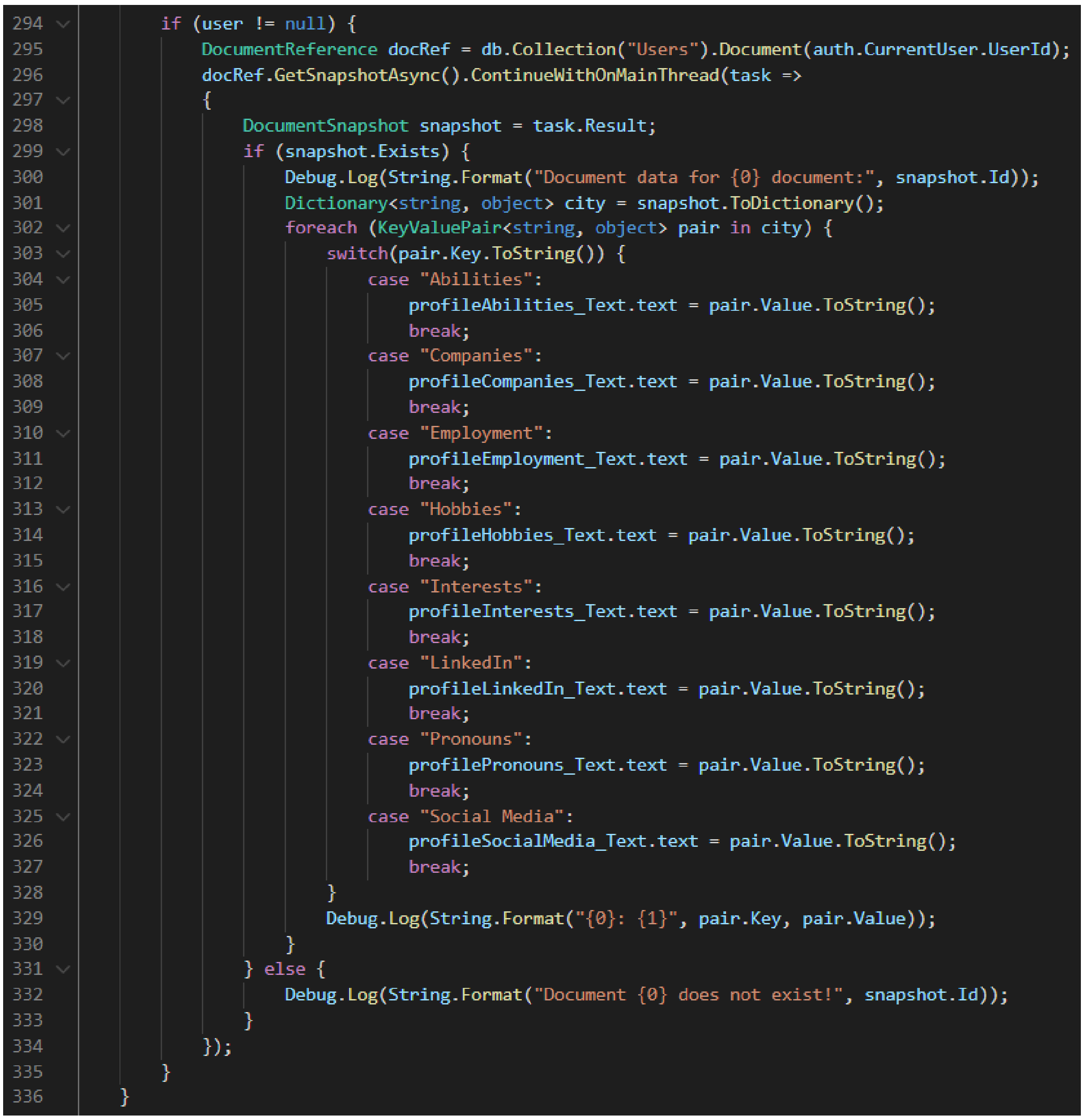
After checking that the profile exists, or after creating their document in the database, a dictionary 'updates' is created (lines 277 – 287) (Figure 3.22) from the user's inputs on the 'Edit Profile' page. The keys are the names for each field in the profile and the key is set as the corresponding user input. This dictionary is then updated under the user's ID in the database (line 289).



```
277         Dictionary<string, object> updates = new Dictionary<string, object>
278         {
279             {"Abilities", profileAbilities.text},
280             {"Companies", profileCompanies.text},
281             {"Employment", profileEmployment.text},
282             {"Hobbies", profileHobbies.text},
283             {"Interests", profileInterests.text},
284             {"LinkedIn", profileLinkedIn.text},
285             {"Pronouns", profilePronouns.text},
286             {"Social Media", profileSocialMedia.text},
287         };
288
289         ProfileRef.UpdateAsync(updates).ContinueWithOnMainThread(task => {
290             Debug.Log(
291                 "Updated the fields of the user's document in database.");
292         });
```

Figure 3.22: Update for dictionary under user's ID in database

Finally, the text on the profile page must be updated to reflect the changes made (Figure 3.23). The user's document is checked again to ensure that it exists in the database and the code loops through each field under the user's ID. A switch expression is used to check the string value of the key and update the relevant placeholder to equal the value of that key. For example, if the code loops through the fields and comes across the key 'Abilities', the placeholder for the user’s abilities on their profile is updated to the value of ‘Abilities’ that is saved in the database.



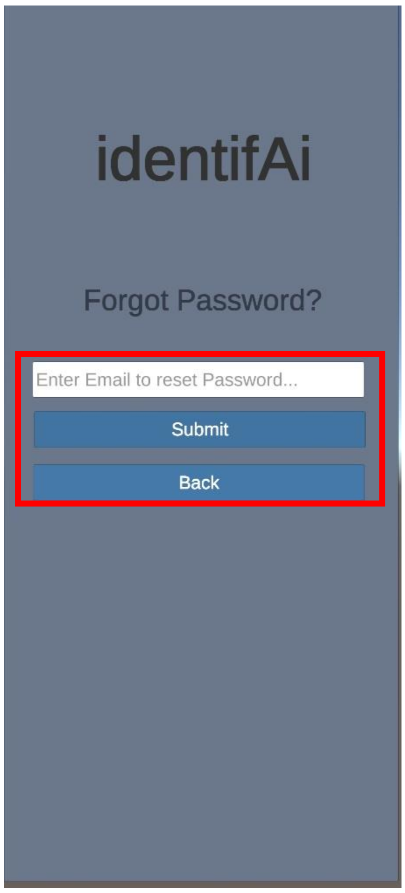
```
294         if (user != null) {
295             DocumentReference docRef = db.Collection("Users").Document(auth.CurrentUser.UserId);
296             docRef.GetSnapshotAsync().ContinueWithOnMainThread(task =>
297             {
298                 DocumentSnapshot snapshot = task.Result;
299                 if (snapshot.Exists) {
300                     Debug.Log(String.Format("Document data for {0} document:", snapshot.Id));
301                     Dictionary<string, object> city = snapshot.ToDictionary();
302                     foreach (KeyValuePair<string, object> pair in city) {
303                         switch(pair.Key.ToString()) {
304                             case "Abilities":
305                                 profileAbilities_Text.text = pair.Value.ToString();
306                                 break;
307                             case "Companies":
308                                 profileCompanies_Text.text = pair.Value.ToString();
309                                 break;
310                             case "Employment":
311                                 profileEmployment_Text.text = pair.Value.ToString();
312                                 break;
313                             case "Hobbies":
314                                 profileHobbies_Text.text = pair.Value.ToString();
315                                 break;
316                             case "Interests":
317                                 profileInterests_Text.text = pair.Value.ToString();
318                                 break;
319                             case "LinkedIn":
320                                 profileLinkedIn_Text.text = pair.Value.ToString();
321                                 break;
322                             case "Pronouns":
323                                 profilePronouns_Text.text = pair.Value.ToString();
324                                 break;
325                             case "Social Media":
326                                 profileSocialMedia_Text.text = pair.Value.ToString();
327                                 break;
328                         }
329                         Debug.Log(String.Format("{0}: {1}", pair.Key, pair.Value));
330                     }
331                 } else {
332                     Debug.Log(String.Format("Document {0} does not exist!", snapshot.Id));
333                 }
334             });
335         }
336     }
```

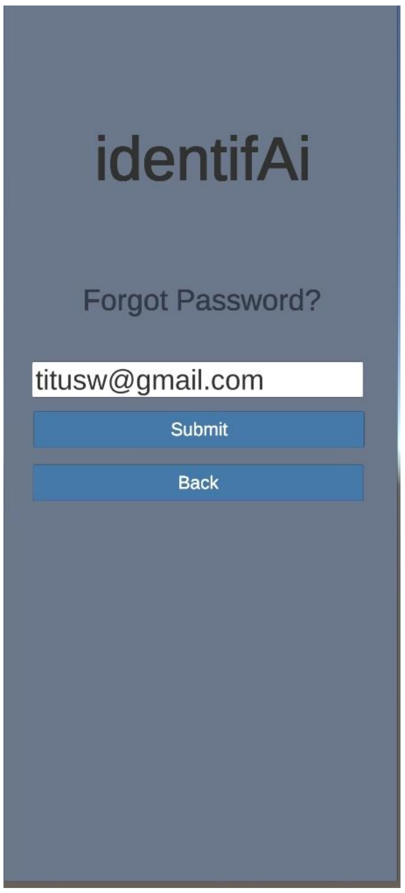
Figure 3.23: Update input on profile page

This code (lines 294 -335) is repeated in the function *Update*, which is constantly running, to ensure that when the user logs in again, the placeholders will read the correct values from the database.

3.3.4 The Forgotten Password Page

The Forgotten Password Page, shown in Figures 3.24 and 3.25 can be accessed through the Login Page through the ‘Forgot Password’ button. This page allows a user to reset their password if they are unable to access their account. Only an email address is required: if an account with that email exists, an email is sent to that account with instructions on how to reset their password. This page is not fully functional yet, as the development team is currently working on the backend to ensure that emails are sent out to the according email addresses.

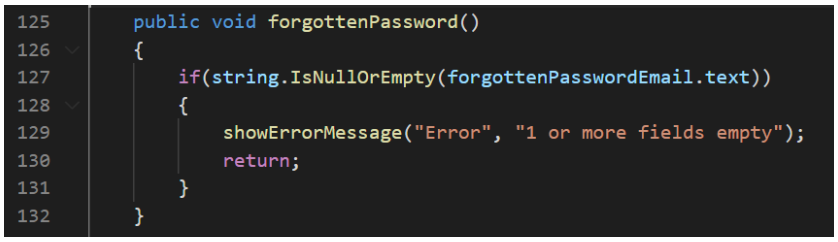




titusw@gmail.com

Figure 3.24 & 3.25: Forgot Password interface without and with example input

The *forgottenPassword* function (Figure 3.26) checks if there is an email address as an input, and if not displays an error message. This function is currently being worked on to implement automatic emails for resetting passwords.



```
public void forgottenPassword()
{
    if(string.IsNullOrEmpty(forgottenPasswordEmail.text))
    {
        showErrorMessage("Error", "1 or more fields empty");
        return;
    }
}
```

Figure 3.26: ForgottenPassword function

3.3.5 Alerts and Error Messages

Alerts can pop up on the user's screen for two different reasons. It will either be a confirmation, for example if an account has been created successfully, or an error with an error message saying what went wrong. Figure 3.27 shows an alert for creating an account successfully, while Figure 3.28 shows the error message for logging in with a wrong password.

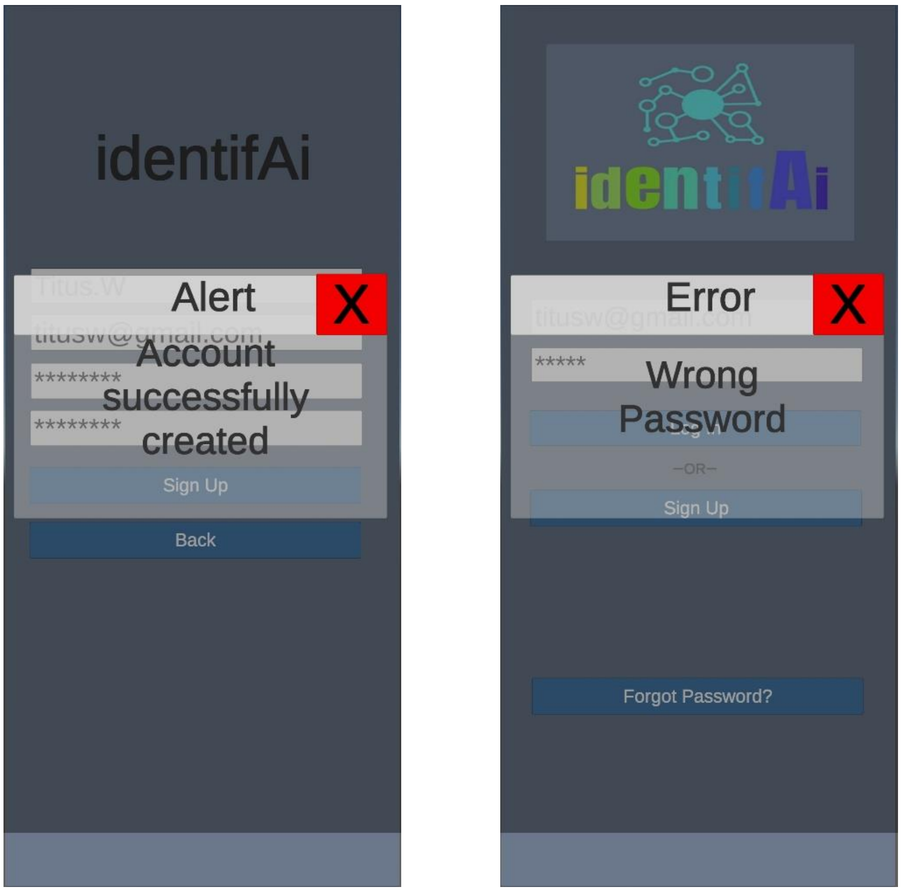
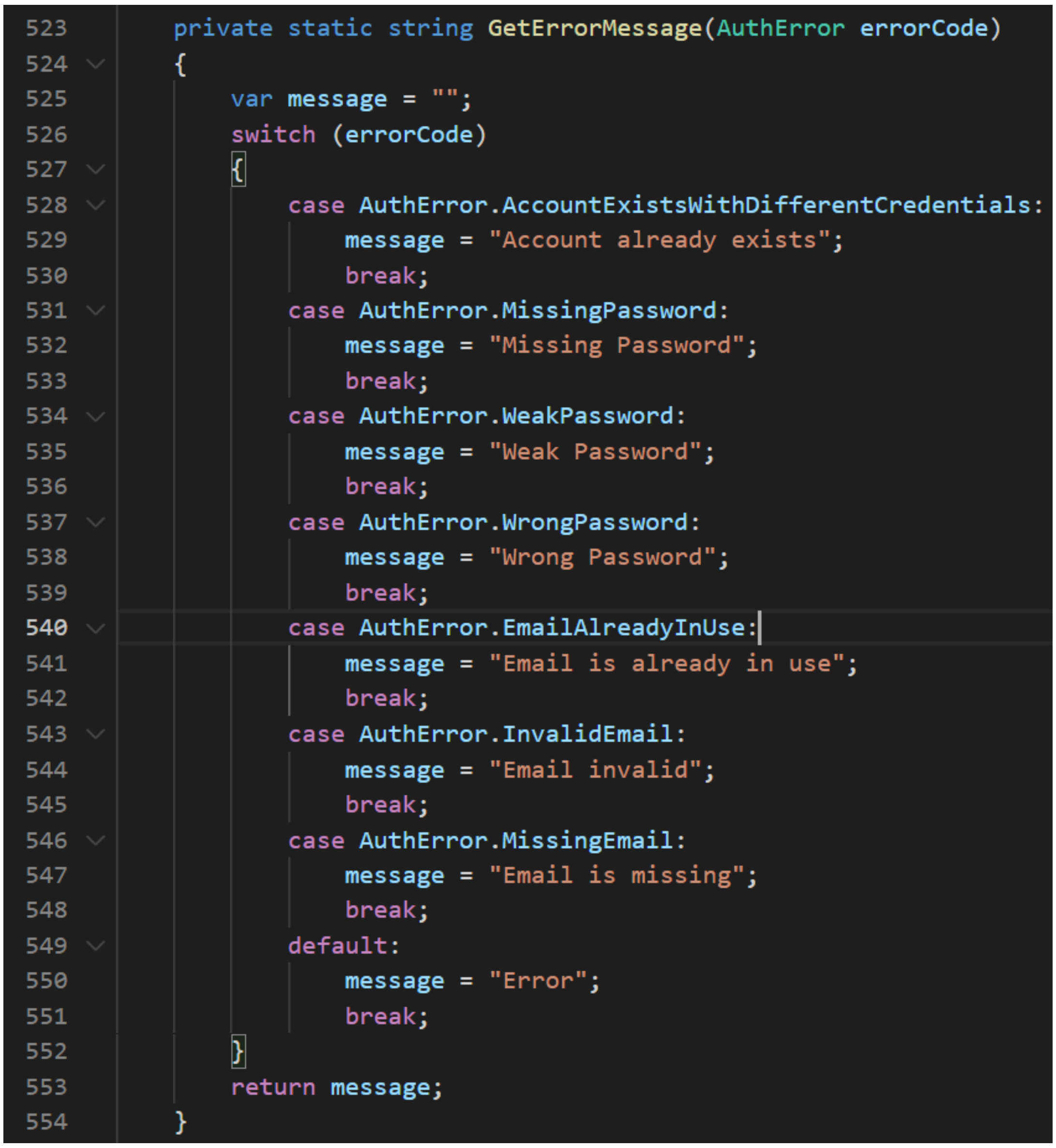


Figure 3.27 & 3.28: Alert and Error messages

The different alerts for confirmations are handled within each specific function, whereas error messages are handled separately in the *GetErrorMessage* function (Figure 3.29). The function uses the switch expression with different cases for each error and displays the matching error message. For example, if the user enters an invalid email address when signing up and submits the input, it is detected by the mentioned function in the back-end and the message “Email invalid” gets displayed to the user. If an error occurs that has no special error case, the default message “Error” is shown.



```
    private static string GetErrorMessage(AuthError errorCode)
    {
        var message = "";
        switch (errorCode)
        {
            case AuthError.AccountExistsWithDifferentCredentials:
                message = "Account already exists";
                break;
            case AuthError.MissingPassword:
                message = "Missing Password";
                break;
            case AuthError.WeakPassword:
                message = "Weak Password";
                break;
            case AuthError.WrongPassword:
                message = "Wrong Password";
                break;
            case AuthError.EmailAlreadyInUse:
                message = "Email is already in use";
                break;
            case AuthError.InvalidEmail:
                message = "Email invalid";
                break;
            case AuthError.MissingEmail:
                message = "Email is missing";
                break;
            default:
                message = "Error";
                break;
        }
        return message;
    }
```

Figure 3.29: Switch expression for handling errors

3.4 AR Scene

The Augmented Reality Page can be accessed by clicking the “AR” button on the profile page and if the app has permission to open the camera, the user can scan their surroundings and find other people that use the app. An example of the user’s screen in the AR Scene is shown in Figure 3.30.

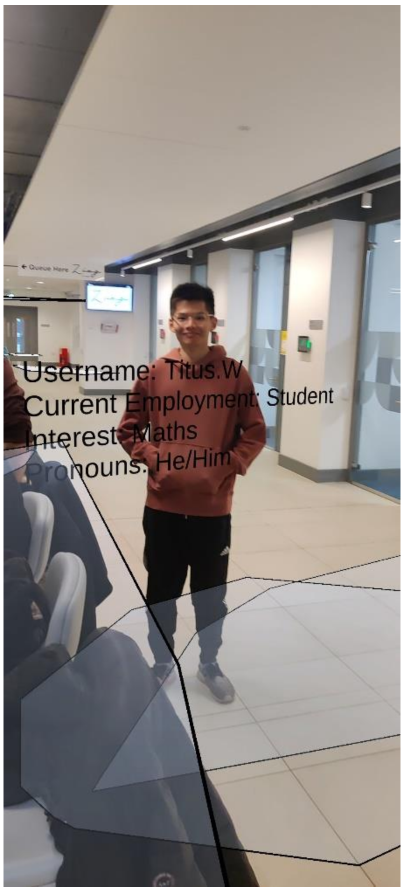


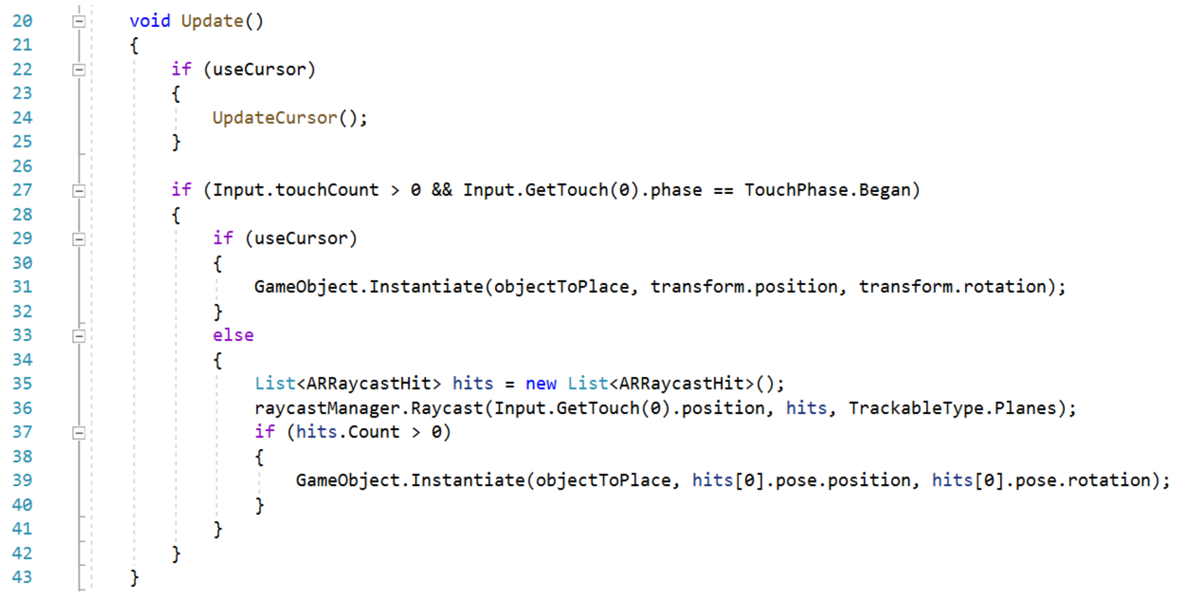
Figure 3.30: AR scene with example output

The relevant files for the AR scene can be found under *Assets/Scenes*.

The ARCamera scene consists of an AR session origin (contained within this an AR camera, which represents the device camera’s location and orientation), AR default plane, AR Cursor game object, and the 3D output text prefab. AR session origin converts a session space to Unity’s world coordinates.

The ARCursor script is applied to the AR Cursor game object to detect the planes (the surfaces of another user) that the user interacts with and to place the 3D text displaying the user’s profile information.

The ‘update’ method is shown in Figure 3.31. Line 27 detects if a user’s action was a touch and not a continuous press. Then, on line 31, place the output text prefab (the user’s profile information) at the current position and rotation of the cursor (corresponding to the camera’s position and orientation), visualising another user’s profile information in the scene (Figure



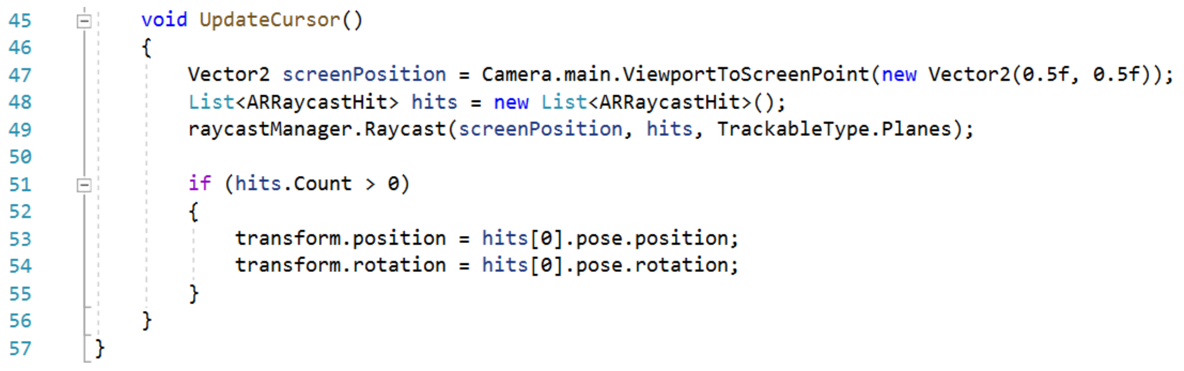
```
20  void Update()
21  {
22      if (useCursor)
23      {
24          UpdateCursor();
25      }
26
27      if (Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Began)
28      {
29          if (useCursor)
30          {
31              GameObject.Instantiate(objectToPlace, transform.position, transform.rotation);
32          }
33          else
34          {
35              List<ARRaycastHit> hits = new List<ARRaycastHit>();
36              raycastManager.Raycast(Input.GetTouch(0).position, hits, TrackableType.Planes);
37              if (hits.Count > 0)
38              {
39                  GameObject.Instantiate(objectToPlace, hits[0].pose.position, hits[0].pose.rotation);
40              }
41          }
42      }
43  }
```

3.31).

Figure 3.31: Update function

To calculate the correct dimensions, line 47 collects the screen position to update the cursor’s position, accordingly (Figure 3.32).

AR plane manager creates game objects for every plane detected in the environment. These game objects create a visible artefact for each plane that is observed. The ARRaycast manager component detects where a ray intersects a trackable (TrackableType.Planes) to determine a hit on line 49 (Figure 3.32).



```
45  void UpdateCursor()
46  {
47      Vector2 screenPosition = Camera.main.ViewportToScreenPoint(new Vector2(0.5f, 0.5f));
48      List<ARRaycastHit> hits = new List<ARRaycastHit>();
49      raycastManager.Raycast(screenPosition, hits, TrackableType.Planes);
50
51      if (hits.Count > 0)
52      {
53          transform.position = hits[0].pose.position;
54          transform.rotation = hits[0].pose.rotation;
55      }
56  }
57  }
```

Figure 3.32: UpdateCursor function

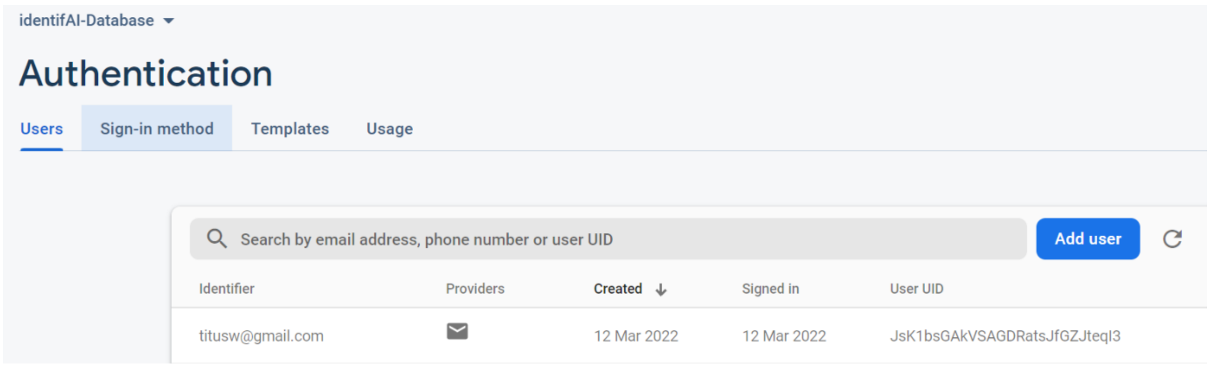
4 Backend Development

4.1 Database management

The database used for this app is *Firebase*, a service owned by Google with many different products and built-in functions which have been used for this project when building the database and linking it to the app in Unity. The database is split up into two different Firebase databases. The first database is *Firebase Authentication*, which is used for login and signing up. The second database is *Firebase Firestore*, which is used for storing the data of individual user profiles.

4.1.1 Firebase Authentication

Firebase Authentication is used for the login and signup process and also supports authentication for phone numbers or popular federated identity providers such as Google, Facebook, or Twitter (these are not currently available in the app but may be added in the future). The database managers can see each user's email address, when the account was created, when the user last signed in and a unique identifier. Additionally, parameters can be hashed to add security and users can be added manually to the database. Database managers also have the option to manually reset passwords or delete and disable accounts. Firebase also offers email address verification, password reset, email address change and SMS verification which can be included in future development. Figure 4.1 shows how the database console in the Firebase website, with the example account.



Identifier	Providers	Created	Signed in	User UID
titusw@gmail.com	✉	12 Mar 2022	12 Mar 2022	JsK1bsGAkVSAGDRatsJfGZJteql3

Figure 4.1: Screenshot Firebase Authentication

The structure of the Firebase Authentication database is displayed in Figure 4.2.

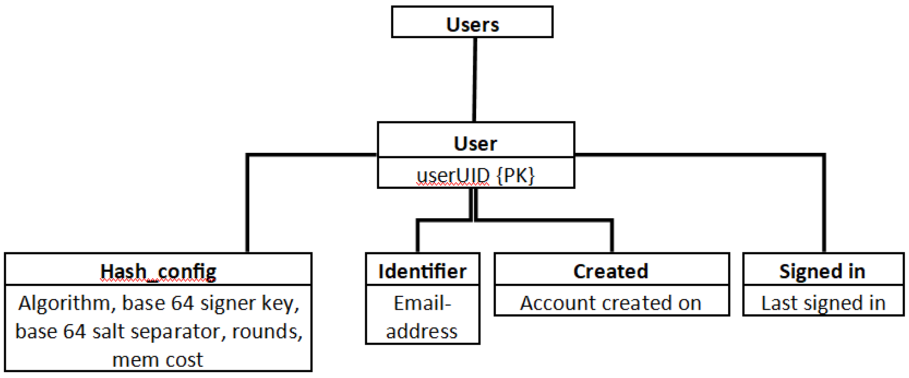
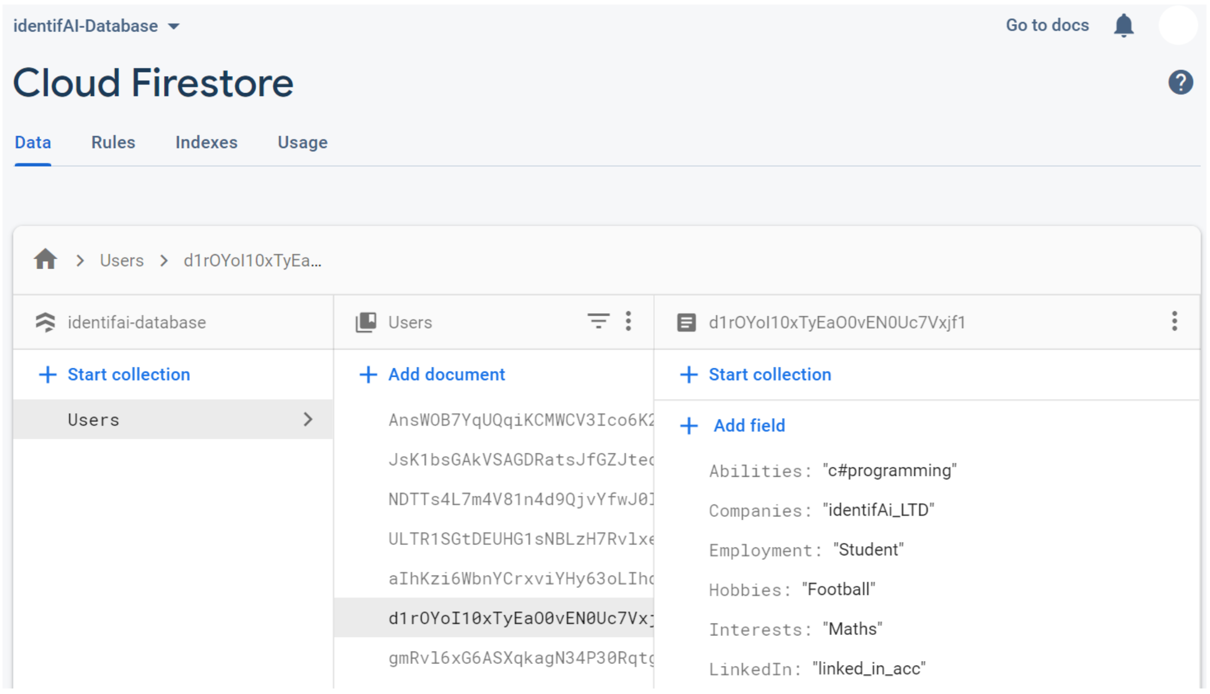


Figure 4.2: Firebase Authentication database diagram

4.1.2 Firebase Firestore

Firebase Firestore is a cloud, NoSQL document database, used in this app to store each user's personal profile, which includes the professional and social information that they wish to share with other users. An overview of the database can be seen in the Firestore console online (shown in Figure 4.3), which can only be accessed by authorised developers.



```
Users
AnsWOB7YqUQqiKCMWCV3Ico6K2
JsK1bsGAkVSAGDRatsJfGZJtec
NDTTs4L7m4V81n4d9QjvYfwJ0l
ULTR1SGtDEUHG1sNBLzH7Rvlxe
aIhKzi6WbnYCrxviYHy63oLIhc
d1rOYoI10xTyEa00vEN0Uc7Vxj
gmRvl6xG6ASXqkagN34P30Rqtg
```

```
Abilities: "c#programming"
Companies: "identifAi_LTD"
Employment: "Student"
Hobbies: "Football"
Interests: "Maths"
LinkedIn: "linked_in_acc"
```

Figure 4.3: Screenshot Firebase Firestore

Currently, the app is only storing the user's profiles in this database. Under the collection 'Users', the document for each user is stored, named by their user ID. In a user's document, the data for the fields of their personal profile is stored: this structure is shown in Figure 4.4.

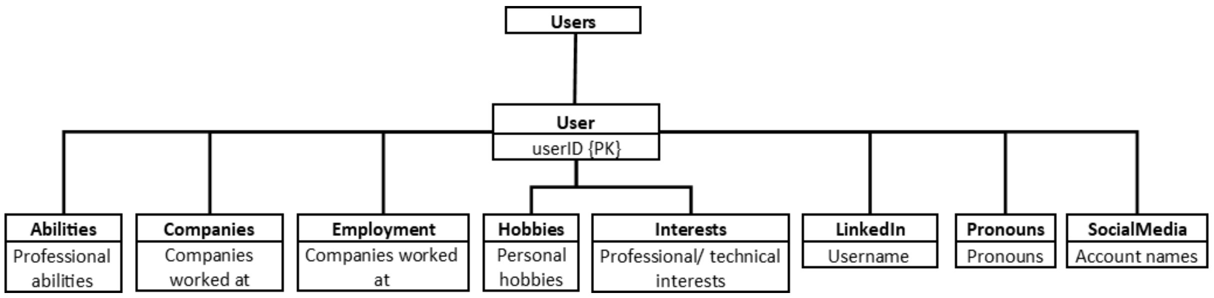


Figure 4.4: Firebase Firestore database diagram

4.1.3 Firebase for App Development

Since the relevant packages are already installed within the project and all required code has been added to the C# scripts, no further setup for Firebase in the project folder is needed. However, to get access to the Firebase console, a developer must connect a Firebase account to the project by requesting permission to the project online from one of the project owners. By emailing one of the following owners, permission can be granted to the Firebase project to access the console:

Andreas Meier: meier.andreas@hotmail.ch

Aelyssa Marquez: aelyssa.marquez@gmail.com

Lara Tolley: laramtolley@gmail.com

The following Firebase account can be used as an example account to access the current Firebase console:

Firebase / Gmail-login: defaultidentifailogin@gmail.com

Firebase / Gmail password: Default+89

5 Troubleshooting

5.1 Contacts

For any problems encountered that are not covered in this document, please contact developers:

Andreas Meier: qnzj81@durham.ac.uk

Aelyssa Marquez: kwsd37@durham.ac.uk

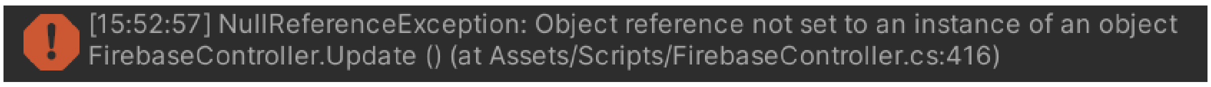
Lara Tolley: kwcq95@durham.ac.uk

Titus Wong: hldl68@durham.ac.uk

Andreas Makris: cdxc13@durham.ac.uk

5.2 Potential Errors

One error that may be encountered in this app's current state is the NullReferenceException Error, such as in Figure 5.1. This error occurs when the app has been built and run, then disconnected by either closing the app or pausing Game mode within Unity. This causes the app to lose connection with Firebase and so the Firestore database cannot be accessed. As a result, after this error occurs, when a user is logged in, since the Firebase database is inaccessible, the profile page will be unable to display the user's details and will also be unable to be edited. To fix this problem, the user must sign out before the app is closed and restarted. Another result of this error is the Android's back button not functioning correctly – when in the AR scene, the back button may not correctly return the screen to the profile page. In this case, the app will need to be closed, then the user will need to login and logout immediately before closing the app again before the app will work.



```
[15:52:57] NullReferenceException: Object reference not set to an instance of an object
FirebaseController.Update () (at Assets/Scripts/FirebaseController.cs:416)
```

Figure 5.1: NullReferenceException Error

Another problem which may be encountered when using this app is having difficulties when detecting an object in the AR scene. This is because AI scene configurations are more likely to detect flat surfaces. Deep learning, a subclass of machine learning in AI that uses neural networks to learn from unstructured or unlabelled data, would be required for more accurate classification using AI facial recognition. However, this may not be feasible due to ethical issues from collection of data without consent, as this would violate protection privacy and internationally accepted ethical norms.

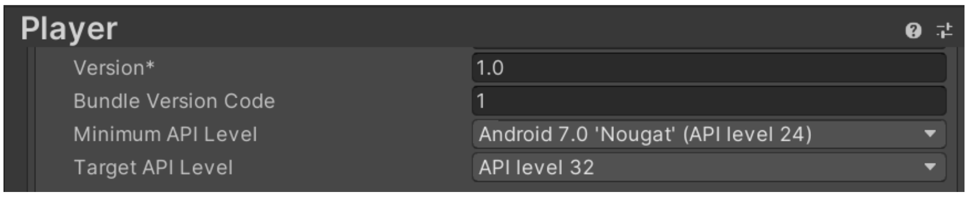
6 Maintenance

6.1 Security

The project can currently only be accessed via the Dropbox link mentioned in Section 2.2. Additionally, the different versions of the app are all stored separately with limited access in case older versions need to be accessed again. Currently only three developers have access to the database, to prevent privilege abuse. Firebase Authentication also uses an internally modified version of Scrypt to hash account passwords. Scrypt is superior in its domain and makes it very expensive for custom hardware attacks to be conducted because it is a memory-hard algorithm. To register an account in this app, the password length required is currently 6 characters although we recommend at least eight characters with one lowercase letter, one uppercase letter, one numeric character and one special character (e.g.,!,@,#). This can be implemented as a requirement for creating an account in future developments of the app, as described in Section 6.2. To ensure accounts remain secure, users should be recommended to never reveal their passwords to anyone else. On top of that, user should also frequently change their password, for example, once a month. Connecting Microsoft Authenticator with our app is currently in development to further restrict third party access to user's accounts (discussed in Section 6.2).

6.2 Future developments

As specified in Section 2.1, the target API levels will continuously be updated to the most recent development to be compatible with the newer generations of Android mobiles and align with Google Play's requirements.


Player

Version*	1.0
Bundle Version Code	1
Minimum API Level	Android 7.0 'Nougat' (API level 24)
Target API Level	API level 32

Figure 6.1: API levels in Payer settings

In order to expand our client base to iOS, some of the app's features will need to be developed using ARFoundation's ARKit package. These changes will be applied to the C# scripts that utilise ARKit's functionality.

Microsoft Authenticator should also be implemented to ensure 2-factor authentication as passwords may be lost, stolen, or hacked, so enabling Microsoft Authenticator for the app will allow users to sign in more securely. Multi-factor authentication ensures that the user proves their identity and helps to prevent people from accessing other users' accounts. In order to implement the Microsoft Authenticator in this app, the Firebase Authentication login system needs to be updated to include authentication using *Microsoft Azure Active Directory*. This can be achieved by following the documentation linked here. Once this has been supported, the sign-in process can also be managed using Azure AD and multi-factor authentication can be set up using the steps outlined in the documentation.

There are several features of the app which have been put into consideration when implementing the app but were not able to be completed due to time restraints. From the requirements specification, the following functional requirements may be added to the app in the future by following the instructions given to implement them:

FR 2.1.1 *A secure login system.* Firebase Authentication currently allows for a simple secure login system, as described in Section 6.1. However, as mentioned, the constraints for accepted passwords could be improved:

At least 8 characters with:

- 1 lowercase letter
- 1 uppercase letter
- 1 numeric character
- 1 special character

An additional function can be added to the Firebase Controller script with conditional statements on the 'create password' input section of the Register page to ensure the password meets the guidelines enforced by identifAI, before allowing the account to be created in Firebase.

FR 2.1.3 : *Camera can scan the room and recognise other app users via GPS, then display their personal information around them in AR.* This functional requirement consists of two different features that can be implemented. The first feature is using the user's GPS location to detect other users in the near vicinity. Unity has a built in function, called *LocationService.Start* which can be used to get the device's location coordinates: the documentation for this function can be found here. By reading the device's location, the app can constantly search for users with a location nearby whenever the AR scene is open, and GPS is enabled. With this first feature working, the second part of the functional requirement can be implemented. The app should be able to display the nearby user's personal information when the camera is pointed at a person in the AR scene. The code in Figure 3.23 can be reused and slightly modified here to fetch the user's information from the Firestore database and displayed within the AR scene.

FR 2.1.4 : *Search Filters (Users should be able to apply filters to reach a certain demographic specifying age, gender, and specific interests).* For this function to be added, a new page of the app will need to be added as a 'Search Page'. Within this page in Unity, the UI should be made up of an input box and a 'submit' button. When the submit button is pressed, a function in the C# script should search the Firebase Firestore database for accounts matching the user's requirements. From this, there is the choice to either display the accounts on this page that meet the search criteria, or instead, only display chosen users' information in the AR scene.

FR 2.1.5 : Login system verifies that user must be 18 or over to create an account (The app should only be accessible to users over the age of 18). This function should not be too difficult to implement in the app in the future. On the 'Sign Up' page, another input box should be added in the UI to take in the user's date of birth when creating an account. In the C# script *FireboxController.cs*, the function *CreateUser* (Figure 3.15) can be modified to only create the account if the date entered is valid (i.e. the date indicates a user over the age of 18). Otherwise, an error message should be generated.

FR 2.1.6 : *Users can link their LinkedIn account with their profile*. This functional requirement also relates to linking social media accounts in a user's profile as the same method should be used for both. This function should allow the user to input a link, then when pressed on a user's profile, open the relevant app on the specified page. In the app's current state, the link is able to be displayed on the profile, but this link is not a hyperlink so is not interactive. One suggestion for this function to work is by using Text Mesh Pro (tutorial linked here).

FR 2.1.7 : *System's ability to disallow any inappropriate words or topics*. A list of any appropriate words can be kept either in the C# file or a separate text file. A function can be applied to every input box where the input is only accepted if none of the words match any words in the list.

FR 2.1.8 : *Users can control what information other users can view about them.* A toggle may be added to the user profile to allow a user to choose whether their profile information or GPS location is shared with other users. This toggle can be added in the UI within Unity, and when enabled, another field (called 'Private') is added to the database under their user ID and set to true. Then within the AR scene, before displaying any nearby user's information, the app should check if this value in the database is set to true: in which case the user's information should not be displayed (otherwise continue displaying information).

FR 2.1.9 *: An AI NLP is used to convert speech to text, and vice versa.* IBM Watson can be integrated into this app to enable natural language processing. This will allow users to convert any text in the app to audio, or if needed, speech to text for search queries. IBM Watson's text-to-speech service provides documentation on their website, as well as their speech-to-text service. The documentation provides details for purchasing and installing Watson for a project.

Another feature which should be updated in future development is adapting the UI to fit different sized screens. Currently, the features in the app are restricted by the size of the screen; overflowing text cannot be seen. The app currently fits in the following resolution: 2400 x 1080 pixels. A scrollbar can be set up in Unity: one way to handle overflowing text is to set up a 'scroll rect' for each field of the profile page. Another way to handle a lot of text in the profile page is to create dynamic textboxes for each field. These textboxes can be put inside a parent container which fills the screen and contains a scrollbar to view overflowing children boxes. In this way, the textboxes for each field will be resized depending on the amount of text inputted by the user.