Mobile Application development Documentation

Isaac Basque-Rice

Table of Contents

[Table of Contents 2](#_Toc3815860)

[1. Introduction 4](#_Toc3815861)

[2. Coding Conventions 5](#_Toc3815862)

[3. Development 6](#_Toc3815863)

[a. Setup 6](#_Toc3815864)

[4. Applying key features and functions of Programming Languages to my app 8](#_Toc3815865)

[a. UI elements, Event Handlers, and Triggers 8](#_Toc3815866)

[b. Data types, operators, constructs and functions 11](#_Toc3815867)

[c. Identifiers and data structures 13](#_Toc3815868)

[d. Programmer defined function or class 14](#_Toc3815869)

[e. The importance of accessing database structures 14](#_Toc3815870)

[f. Usage of a complex third-party function or class 15](#_Toc3815871)

[5. Development Tools 21](#_Toc3815872)

[a. Integrated Development Environment (IDE) 21](#_Toc3815873)

[b. Installation and Configuration 25](#_Toc3815874)

[c. Emulators and physical devices 28](#_Toc3815875)

[6. Creation and deployment of a working mobile app 34](#_Toc3815876)

[a. Appropriate features and functions 34](#_Toc3815877)

[b. Debugging tools and their uses 48](#_Toc3815878)

[c. Deployment to Virtual Device and Hardware Device 49](#_Toc3815879)

[d. Review of mobile app against client spec and customer feedback 59](#_Toc3815880)

[e. Evaluation of app from Client feedback and update and maintenance strategy 60](#_Toc3815881)

[7. Documentation 61](#_Toc3815882)

[a. User Documentation 61](#_Toc3815883)

[b. Technical Documentation 65](#_Toc3815884)

[References 73](#_Toc3815885)

# Introduction

The purpose of this document is to serve as a record for the process of developing an application for the Studio School Liverpool (Henceforth referred to as ‘Client’, or ‘The Client’), a school for 14-19-year-olds based in the centre of Liverpool. This application should serve a few specific functions, firstly, and most importantly for the overall purpose of the application, it should allow a user to register an account with the school, allowing for the input of a first name, surname, username (Possibly to become optional), password, and email address for contact purposes. Additionally, as laid out in the design document that precedes this, the colour scheme of the Studio School must be adhered to, as well as branding and overall look of said application. This is the defined user need for the application.

Prior to the actual development of the application several aspects must be touched upon, these aspects are as follows

* Coding conventions, that being the conventions I must adhere to in order for my code to be coherent to myself and, indeed, those assessing my work and, in a wider context, those seeking to build upon it in the future. These conventions include frequent commenting, explaining what code snippets do etc., keeping file names in the same format (Camel casing etc.) and the same with variable names, this includes, camel casing and appending a prefix that explains what the variable does.
* Development tools, everything contained within the write up section of the documentation after development tools falls under this category, this section will describe the most essential and noteworthy development tools I will be making frequent use of specifically in the process of developing this application, these tools include the IDE and the configuration and features of the IDE, as well as the ability to run the application off of a physical hardware device (mobile phone).

# Coding Conventions

As mentioned above, coding conventions are a fundamental requirement when one is programming anything, from a calculator to a fully featured application and anything in-between. The reason for this is so that any developers don’t get confused when viewing or editing code. For example, if default names are given to every button (button1, button2, etc.) this will cause large amounts of confusion from a development context due to the fact that developers won’t know instantly what a button does and therefore have to exert themselves unnecessarily by committing numbers, and their relations to actions within an application, to memory. In my application I will get around this by adding a descriptive prefix to the beginning of my variable names, btn\_X for button, tb\_X for text box and so on. Additionally, I will also add a descriptive name beyond the prefix (represented above by Xs), for example a submit button would be described by the name btnSubmit, or a first name text box would be described by txtFirstName.

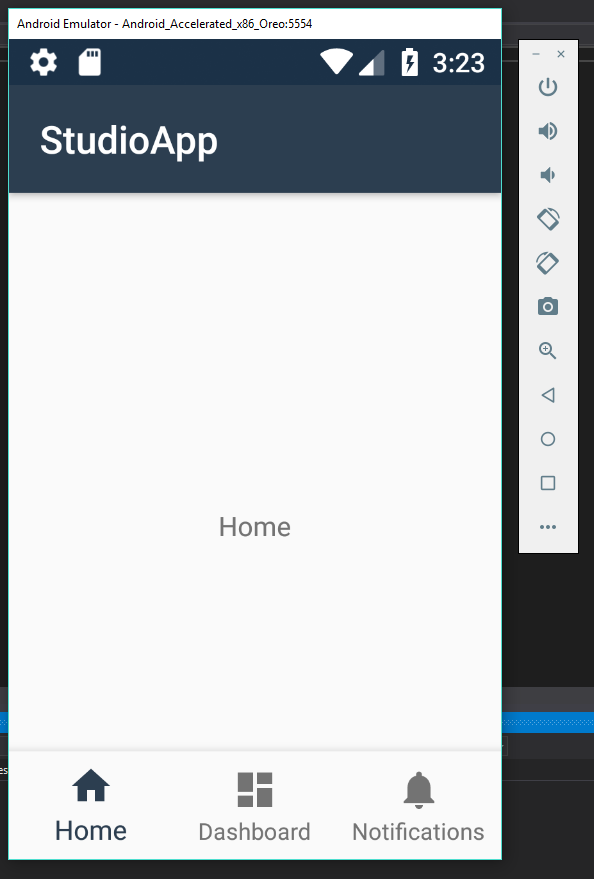
Another coding convention that I must adhere to is also demonstrated in my txtFirstName example, that is what is known as Camel Casing, this is wherein phrases that would usually be split up by a space (Such as ‘FirstName’ in the above example) are instead not split up, but rather the two words are distinguished from each other by the sue of capital letters. Camel casing is almost exclusively used within the context of naming conventions, such as file names and variable names, this is due to the lack of ability to use spaces in programming as a general rule.

Yet another imperative convention one must adhere to is code commenting, this is where sections of code are explained through an in-IDE commenting method. Within programming languages (as opposed to Mark-Up languages) comments are usually discerned by the use of a double backslash (‘//’) (Mark-Up languages use a different system, allowing users to discern the beginning of a comment by typing ‘<!--’ and ending a comment with ‘-->’) at the beginning of a comment line, as well as a different colour of text as to be more visible to the reader (Green in the case of Visual Studio). The reasoning behind code commenting is fairly obvious, it is to explain sections of code, what they do, why they’re there etc. in order to streamline the coding process somewhat, and also, in an assessment context, to show what I have been doing and what my process is.

# Development

## Setup

Below is a screenshot of the first successful run of any application relating to this project within an emulator, you may notice the distinct lack of client-based design, this is due to the fact that this application is a prebuilt one and as such is designed to fit many purposes, and, in my case, will serve as a base for design of the application generally.



Screenshot 1, the earliest form of my applucation as ran in an Emulator

A distinction the final app will have from the original design as outlined in a previous document is that the navigation terminal will be located at the bottom rather than the top, this is for ease of use due to the fact that the way that users tend to hold mobile devices allows for access to the bottom of the screen much easier than the top.

# Applying key features and functions of Programming Languages to my app

## UI elements, Event Handlers, and Triggers

I have included in this app a varying amount of UI elements and views, including but not limited to ScrollView, forms, and buttons, which each serve their own purpose in the context of the application.

Firstly, ScrollView, as is implied by the name, allows the user to scroll down a screen, this view is implemented, obviously, in solutions wherein the application has too much content to fit onto a single screen.

The following code is implemented as a parent of the <LinearLayout … /> tags and allows a user to scroll through elements on a screen

<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"

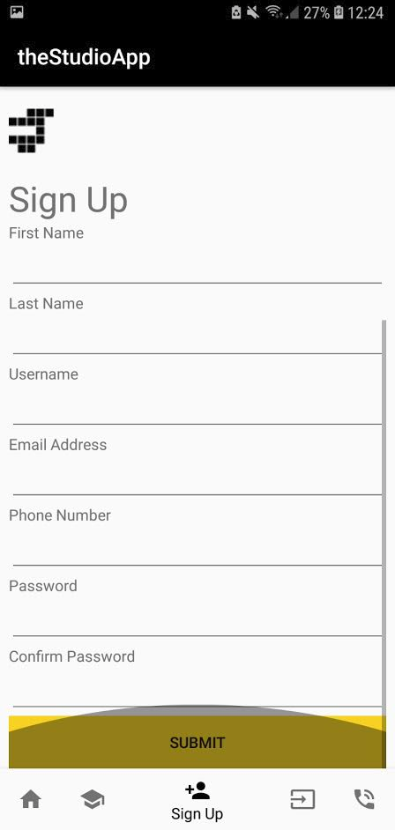
android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<Content goes here>

</ScrollView>

Next, we have forms, which allow the user to input their own personal information and commit it to some form of database through the usage of SQL and an event handler, usually a button. Below is an example of a form, the sign-up form I have in my application:

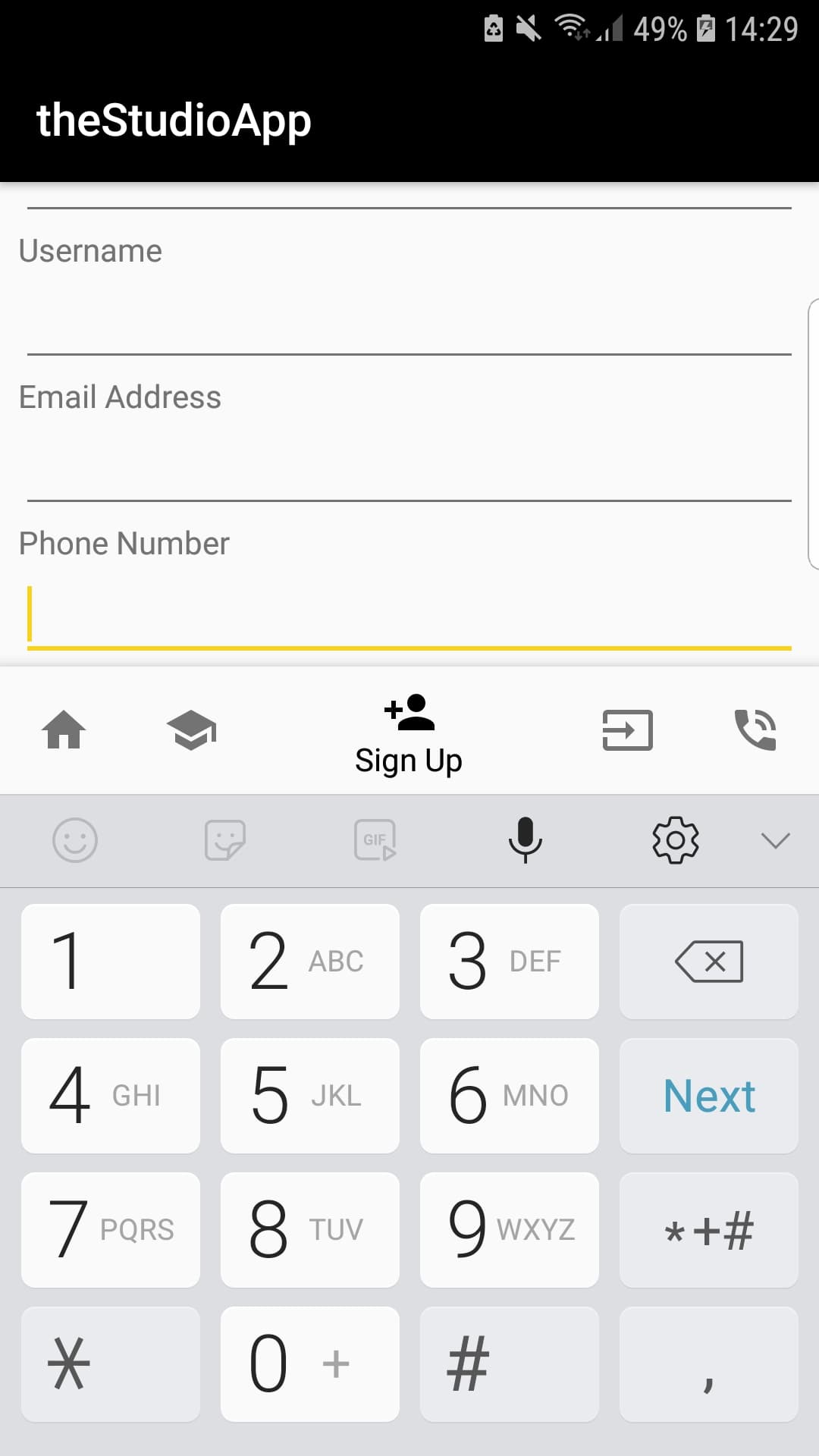


Screenshot 2, the sign up tab featuring new submit button and Studio logo, also note the grey dome, this shows that the bottom of the screen is reached in scroll view

Below is a table describing all UI elements in this screenshot:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Control | Properties | Name | Handlers | Purpose |
| Button | Font: Roboto  Text Size: 42  ForeColour: #f7d31d  BackColour: #000000 | btn\_Nav (when referred to as a group of 5) | OnClick | Switch from screen to screen between my application |
| Textbox | Font: Roboto  TextSize: 48  Colour: #000000 | txtFirstName,  txtLastName,  txtUserName, | N/A | Allows user to input their first name, surname, and username respectively |
| Textbox | Font: Roboto  TextSize: 48  Colour: #000000 | txtEmailAddress | N/A | Allows user to input email, has an email validation regex statement:  “^\w+([-+.']\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*$” |
| Textbox | Font: Roboto  TextSize: 48  Colour: #000000 | TxtPhoneNumber | N/A | Only accepts int inputs |
| Textbox | Font: Roboto  TextSize: 48  Colour: #000000 TextReplace: Password | txtPassword | N/A | Replaces all character with a dot character, allows users to input passwords |
| Button | Font: Roboto  TextSize: 42  ForeColour: #f7d31d  BackColour: #000000 | btnSubmit | OnClick | Allows users to enter their personal information into the database |

UI elements in this screen come in two forms. Firstly, text boxes, which come in sub categories such as "textPersonName", "textEmailAddress", "phone", and "textPassword", which have their own parameters, for example in "phone" only int data types are allowed, as such a number pad appears when one clicks on it as opposed to the standard QWERTY keyboard, like so:



Screenshot 3, the Phone Number text box with the number pad displayed

Additionally, a submit button is displayed in Screenshot 26, at this point it is not functional however upon the creation of the backend (SQLite code and prerequisite files), this button will be coded to serve as an event handler and commit the inputted data to a database.

## Data types, operators, constructs and functions

As far as data types are, a fundamentally clear application of these are in the previously mentioned forms, specifically strings and integers, with an obvious preference being placed upon strings in this instance due to the need to input names, passwords, emails etc. In addition to text editing fields in the solution, the un-editable text elements, for example the titles of each screen that display towards the top, and the lorem ipsum that can be found throughout the application, can all be found in a file known as strings.xml which is entirely dedicated to storing text for later reference throughout the solution.

Yet another data type I make use of is ‘bool’ or the Boolean data type, which describes an object that can exist in one of two states, usually denoted by true and false but also occasionally described by on and off or 1 and 0, respectively. Specifically, I use it in the previously described context of fragments, wherein five individual fragments are stored in the root file of my solution and referenced in MainActivity.cs through a Boolean class. This class’s purpose is to generate and switch to each pre-defined fragment once the corresponding button is pressed, for example if I press the ‘home’ button in the navigation this class will generate the page and return true. The following is the Boolean class:

public bool OnNavigationItemSelected(IMenuItem item)

{

switch (item.ItemId)

{

//if a user navigates to screen x create fragX and switch to it

case Resource.Id.navigation\_home:

SwitchToFrag(new fragHome());

return true;

case Resource.Id.navigation\_aboutUs:

SwitchToFrag(new fragAboutUs());

return true;

case Resource.Id.navigation\_signUp:

SwitchToFrag(new fragSignUp());

return true;

case Resource.Id.navigation\_logIn:

SwitchToFrag(new fragLogIn());

return true;

case Resource.Id.navigation\_contactUs:

SwitchToFrag(new fragContactUs());

return true;

}

return false;

}

With regards to operators, it is easy to not even consider them most of the time when one is coding, these are +, -, /, \*, =, >, <, and many more. A code example of where I have implemented an operator is below, with the operator(s) highlighted.

SignUpTable user = new SignUpTable

This line instantiates an object of the SignUpTable and makes use of the = operator to declare that the object, SignUpTable, called ‘user’, equal to a new instance of the SignUpTable object.

## Identifiers and data structures

Firstly, Identifiers serve to, as the name suggests, identify classes, variables, file names, and other objects in code. All identifiers must stick to a set of rules enforced as good coding practices and standards, for example all identifiers must contain only alphanumeric characters (A-Z, 0-9, with the exception of the underscore and dash characters (\_ and - respectively)), it cannot be the same as a phrase used in code (for example bool, string, int, etc.), and cannot under any circumstances include spaces, instead camelCasing or a similar method of designating separate words (such as with dashes or underscores) must be used, below is an example of a good practice:

namespace theStudioApp

Next, data structures are data organisation tools that allow for efficient storage and data management in a specific context, an example of this in general computer science would be an array, which is a number of elements, typically of the same data type (most notably the int data type) in a specified order. Applied in our context, data structures will most notably and heavily be used in data storage, clearly, with SQLite making use of a self-contained and streamlined version of the standard Relational Database Management System, wherein each user is set an individual ID number and from this users can be stored in an array where only their ID is actually stored in said array and references the rest of the user data elsewhere in the RDBMS itself.

In a standard context, that is, for example, a website, a data structure in the form of an RDBMS would exist externally in a separate hosting provider and a specialised, dedicated piece of software, this is not the case in this app, we use SQLite, a lightweight data structure that is directly integrated into the application, meaning no data is stored off device, a connection to the internet is not required, security is preserved as nobody without physical access to the device can access it, and very little is lost in the way of functionality for the user.

The importance of data structures like this cannot be overstated, the ability for a user to access their personal information is of the utmost importance, and being able to store a secure username/password combination on-device like this is a godsend.

## Programmer defined function or class

Continuing on with the theme of SQLite my ‘programmer defined function or class’ will, in this instance, define the required fields within the database, these will be the as yet already mentioned ID field, which serves as the primary key from which the rest of the data in the database pertaining to that specific user will be referenced, as well as the first and last names, username, email address, phone number, and password

public class SignUpTable

{

[PrimaryKey, AutoIncrement]

public int ID { get; set; }

public string firstName { get; set; }

public string lastName { get; set; }

public string username { get; set; }

public string emailAddress { get; set; }

public int phoneNumber { get; set; }

public string password { get; set; }

}

This meets the user’s need to log in and sign up to the application.

## The importance of accessing database structures

The importance of accessing database structures is self-evident, accessing and being able to modify these structures is critical to a functioning sign up and login system due to the fact that users will not be able to even create an account, let alone sign up for one. The creation of a user account is vital to the specification of the app primarily due to data collection methods, in addition it’s not beyond the scope of possibility that dynamic application content may be added at some point, meaning the appearance of the app, displaying a user’s name and “Hello [firstName]”, for example, will be predicated on the ability for a user to create said account, which, without access to the database’s data structure, is impossible.

Further to the point of log-in and sign-up functionality, the ability to authenticate users against a database structure through accessing said structure is of course of the utmost importance for a number of reasons. Firstly, the user’s ability to get into their own account gives them access to any amount of information they have provided or specified, for example in the future the app’s functionality may be expanded to show timetables or a teacher’s class table, which is dynamic and specified to the individual, and as such would require the implementation of accounts.

Next, and arguably the clearer argument for the importance of accessing database structures, is the level of security and data integrity it affords. Let us work off of the previous example and assume a significant amount of employee or student details were within the app, without a secure login functionality this information could very easily be accessed by anyone with the app and exploited to who knows what end. Of course, user accounts and the ability to access these database structures aren’t perfect, but they allow for a crucial safety barrier to be in place that would not be there otherwise.

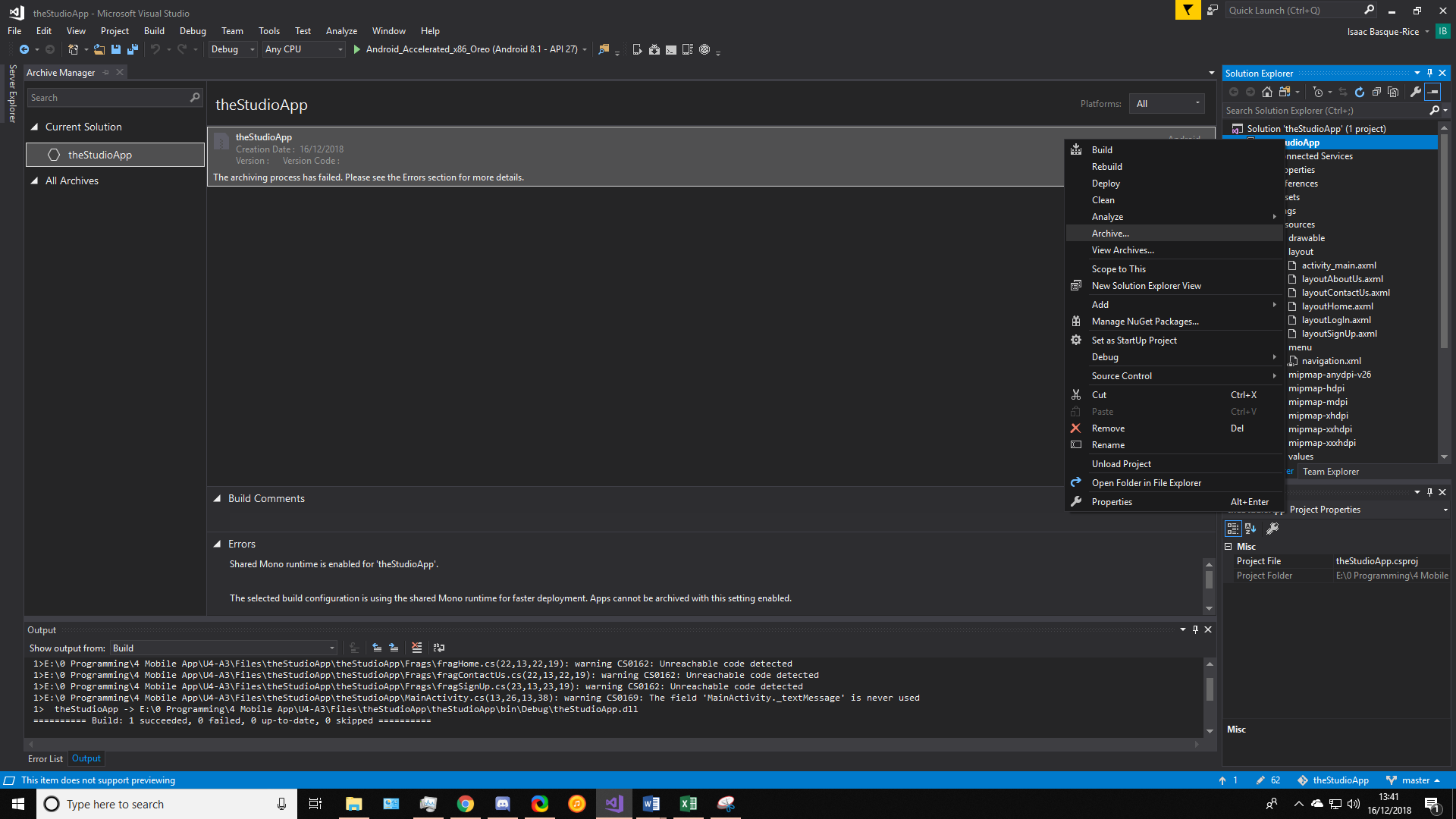
## Usage of a complex third-party function or class

Upon the publication of an application on any platform of note in the modern mobile industry, what is known as ‘code signing’ is an absolute requirement. Code signing is a practice that allows the end user to be 100% assured that the code contained within a downloaded application is totally malware free and devoid of security flaws of any kind. Additional benefits to code signing an application include the fact that the developer themselves are credited, in a similar way to how a painter may sign their painting on completion, as well as the ability to update their application with relative ease (Android Source, 2018). With this in mind it becomes apparent that, if I were to publish an app on the Google Play store I would need to make use of Google’s proprietary third-party code signing certificate, a ‘complex third-party function’.

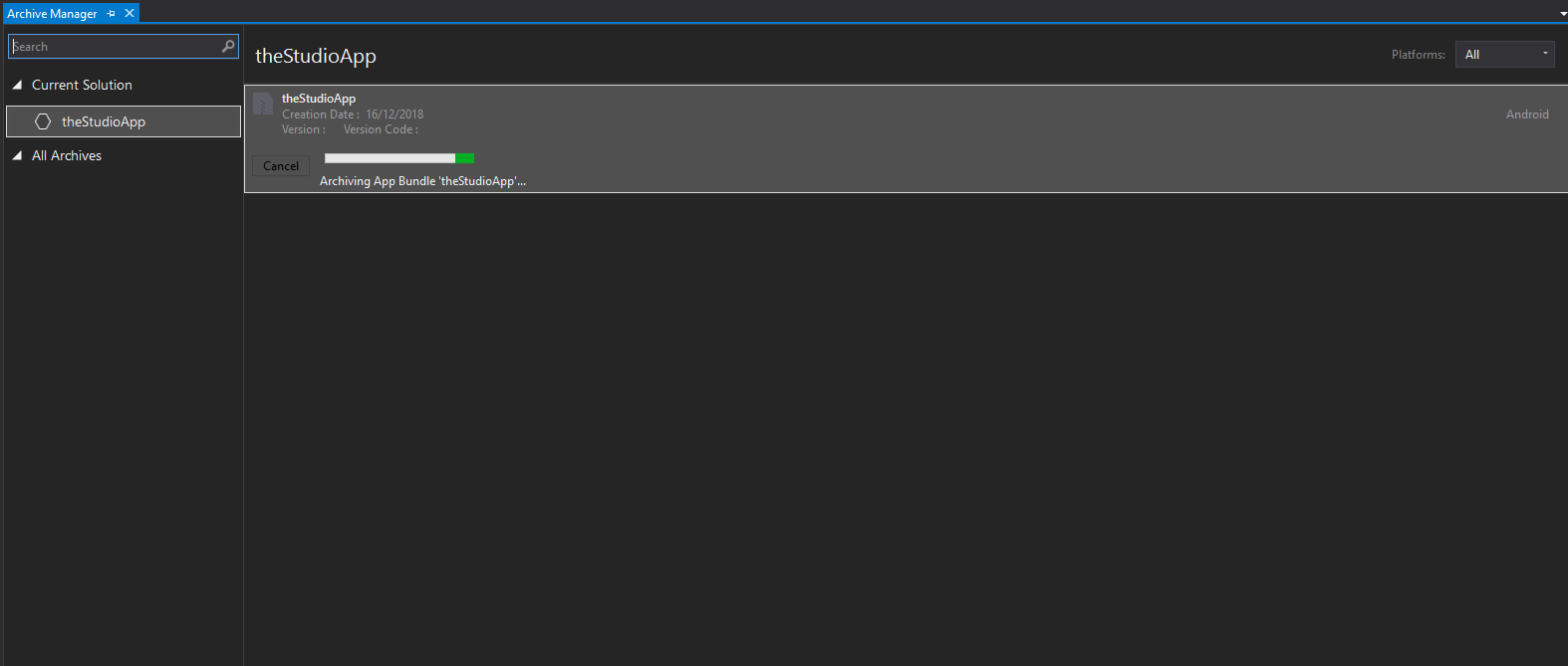
App signing works using a certificate and key system, the latter of which is fairly similar to encryption. The signing process entails the generation of a certificate and a public key, represented by a fairly long prime number, the certificate contains this public key as well as a fairly significant amount of metadata such as name and location, this is to ensure that the application comes from a reputable source (i.e. a confirmed legitimate developer i.e. myself, in this instance), This is also the same method that allows updates to be released with such ease. The owner of the certificate (once again in this instance, myself) also holds the private key, this ensures the public key used by the user is legitimate and so the download of the application can go ahead. (Android Developers, 2018)

The certificate itself is held in what is known as a keystore, this keystore will also contain the metadata in the certificate such as name, organisation, and location. Keystores also allow for their validity to be specified in terms of length of time, so for example, one may wish for their keystore to be valid for 30 years, this allows and Android APK file to be signed with valid information every year for 30 years. (McLemore, et al., 2018).

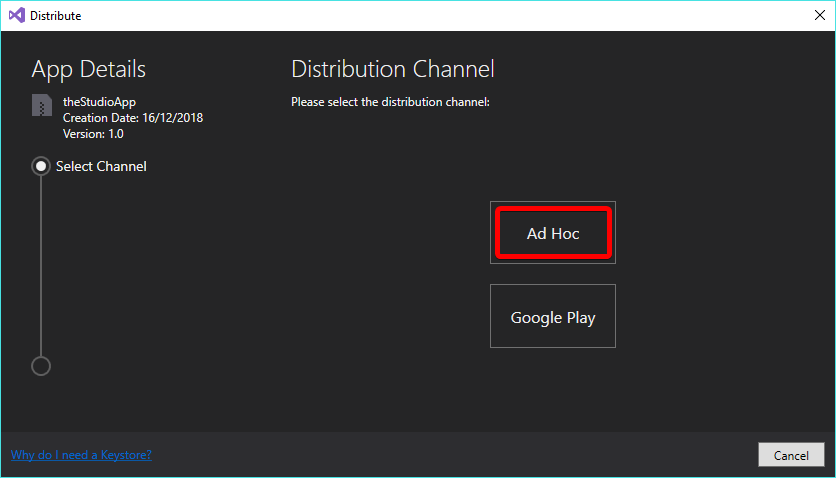
In Visual studio the process is as follows:

1. Right click the project in the solution explorer and click ‘archive’: 

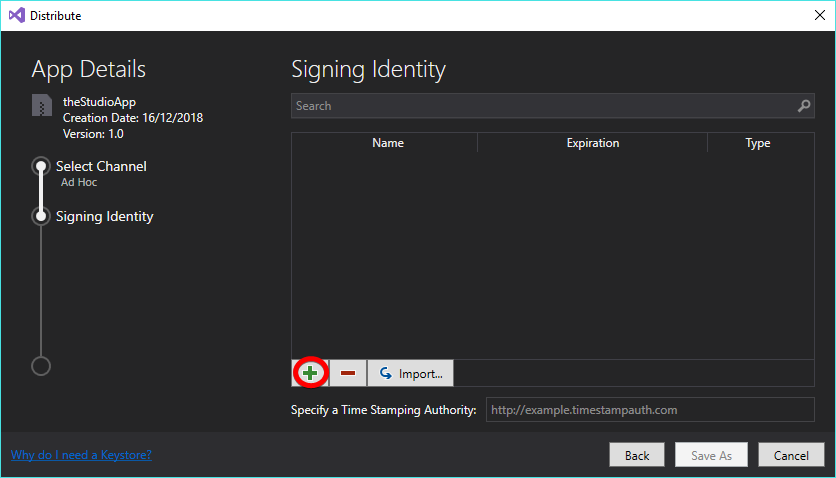
Screenshot 4, the right-click menu for theStudioApp solution with Archive highlighted

1. The archive manager will appear and begin archiving the project as such:

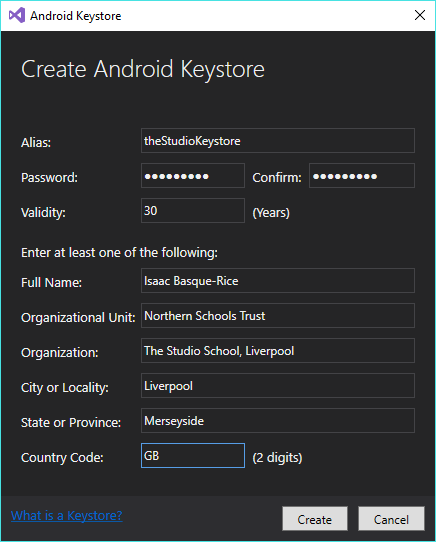
Screenshot 5, the Archive Manager for theStudioApp

1. After clicking the distribute button in the bottom right of the archive manager the distribution menu appears, click the ad hoc button: 

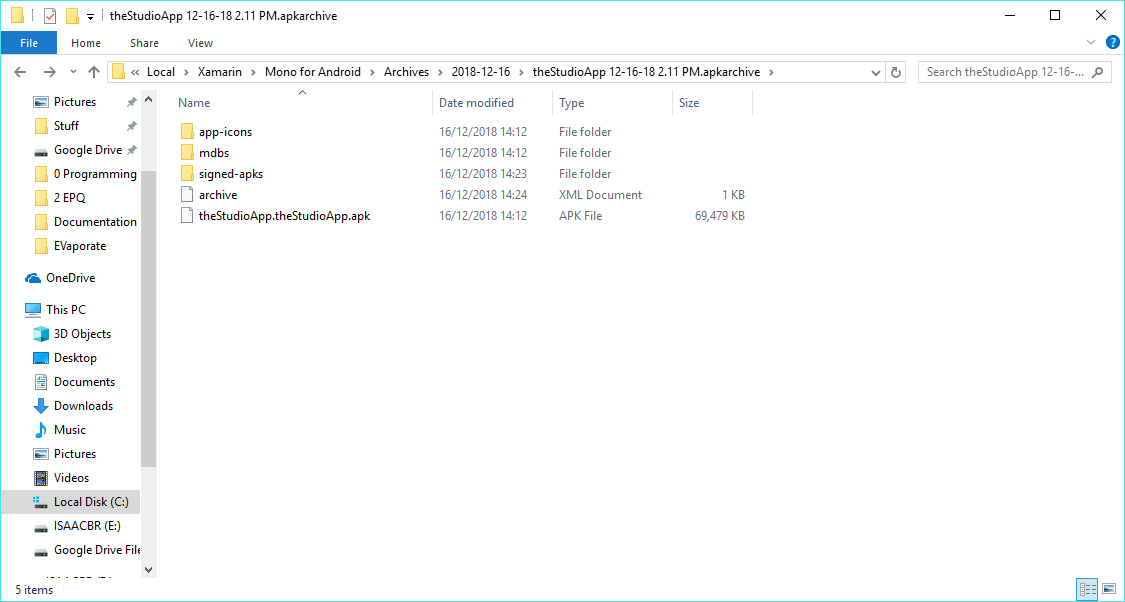
Screenshot 6, the distribution menu

1. The creation of the keystore comes next, press the green + button to create a new one

Screenshot 7, the second layer of the distribution menu, the code signing menu

1. The filled in keystore creation menu is displayed below, this is the template from which the keystore will be formed

Screenshot 8, the third layer, the form one must fill out to create the keystore

1. The keystore has now been created (after a prompt asking for a password) and is stored in location C:\Users\[user]\AppData\Local\Xamarin\Mono for Android\Archives\2018-12-16\theStudioApp 12-16-18 2.11 PM.apkarchive

Screenshot 9, the file location for the keystore

# Development Tools

Next, we have tools that will aid me in the process of developing the application, these include the IDE, emulators, SDKs, plugins, and all other aspects of the application that I am making use of that were not developed by myself.

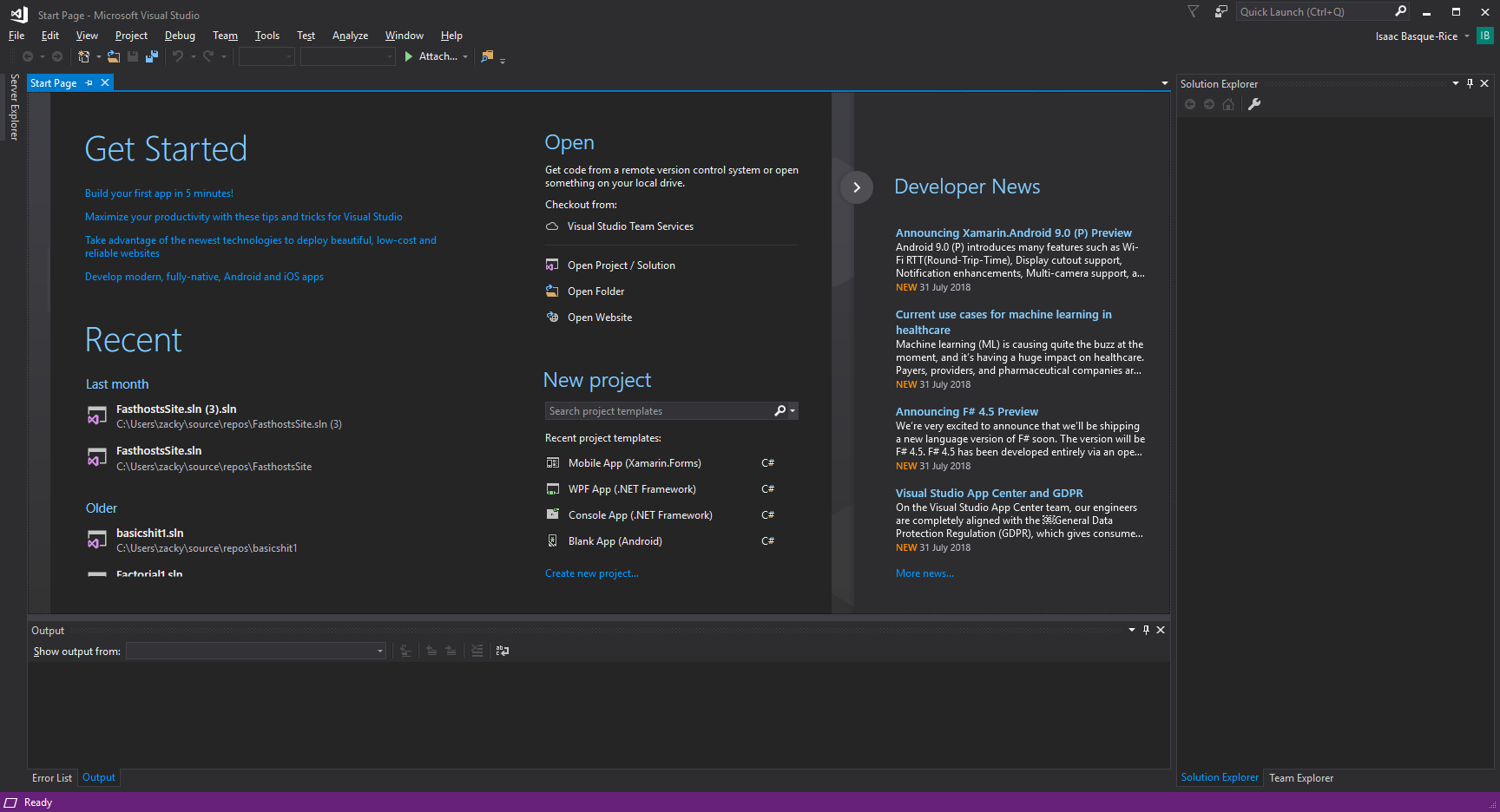
The tools I will be talking about in this section fall under five distinct categories, these are shown below, and in addition to this I will be using the Xamarin cross platform development suite which allows for mobile application development across all three major operating systems and deployment platforms, these being Android, iOS, and UWP. As explained in a previous document, however, I will not be making use of Xamarin’s full suite, opting instead to focus my efforts solely on Android development, this is due to several reasons, most notably that the publishing stage is more intensive and requires me to own hardware I don’t have, as well as this I find it will provide an unnecessary challenge within development and will not have the same support infrastructure as Android will.

## Integrated Development Environment (IDE)

The Integrated Development Environment, or IDE, I intend on using for this project, and indeed most projects in the foreseeable future, is Microsoft’s Visual Studio. This IDE has flawless integration with the Xamarin platform and allows the use of the C# programming language (in which I am most proficient) natively. Both of these facts are due to the fact all three (VS, C#, and Xamarin) are owned, operated, and maintained by Microsoft, and therefore integration of the three is seamless.

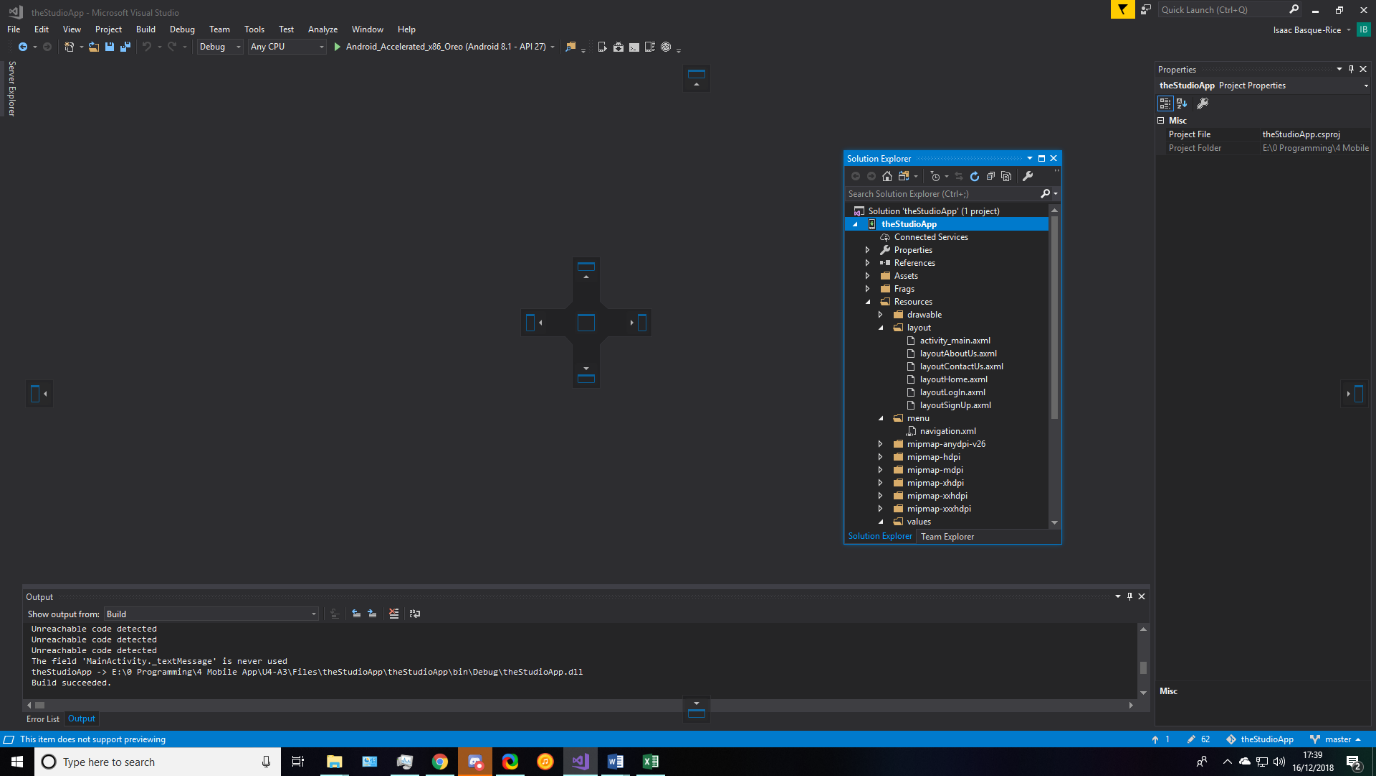
In addition to this, and more generally, the Visual Studio IDE is entirely fully featured, featuring a WYSIWYG (What You See Is What You Get) editor and a large number of tools for using this, including drag and drop text boxes and buttons. As well as this, and most importantly of all, development of anything using this IDE comes at 0 cost to the developer, meaning any individual with a sufficient computer can program using it.

Below is a screenshot of the opening screen of the Visual Studio IDE



Screenshot 10, the Visual Studio landing screen

As you can see from the screenshot below, Visual Studio offers A large degree of customisation when it comes to the layout of the environment, note how the window that is detached can be placed at any position on the screen at essentially any size, and can be snapped to any side via the use of the cross-like object centre-screen:



Screenshot 11, An example wherein the solution explorer window is detached from its normal resting position

Microsoft is well noted in development circles for multiple methods to achieve anything at all, for example in word one is able to shift from normal text to bold by making use either of the ‘ribbon’ at the top OR a keyboard shortcut, in this case, ctrl+B. This design principle permeates the entirety of the Microsoft software suite, from office products such as Word, Excel, and PowerPoint, to development tools such as SQLServer and, of course, Visual Studio.

What follows is a series of actions that one can take in the Visual Studio IDE in numerous ways.

Debugging, the process by which a program in development can be deployed/ran so a developer can fix bugs within said program, this can be achieved on the ribbon as seen below, by pressing the F5 button on the keyboard, or by going into the ribbon menus at the top and clicking Debug > Start Debugging



Screenshot 12, The debug button on the ribbon

Undoing and redoing, restoring a program to a previous state, can be done by pressing Ctrl+Z and Ctrl+Y on the keyboard, using the ribbon shortcuts as seen below, or by going to Edit > Undo/Redo in the menus at the top

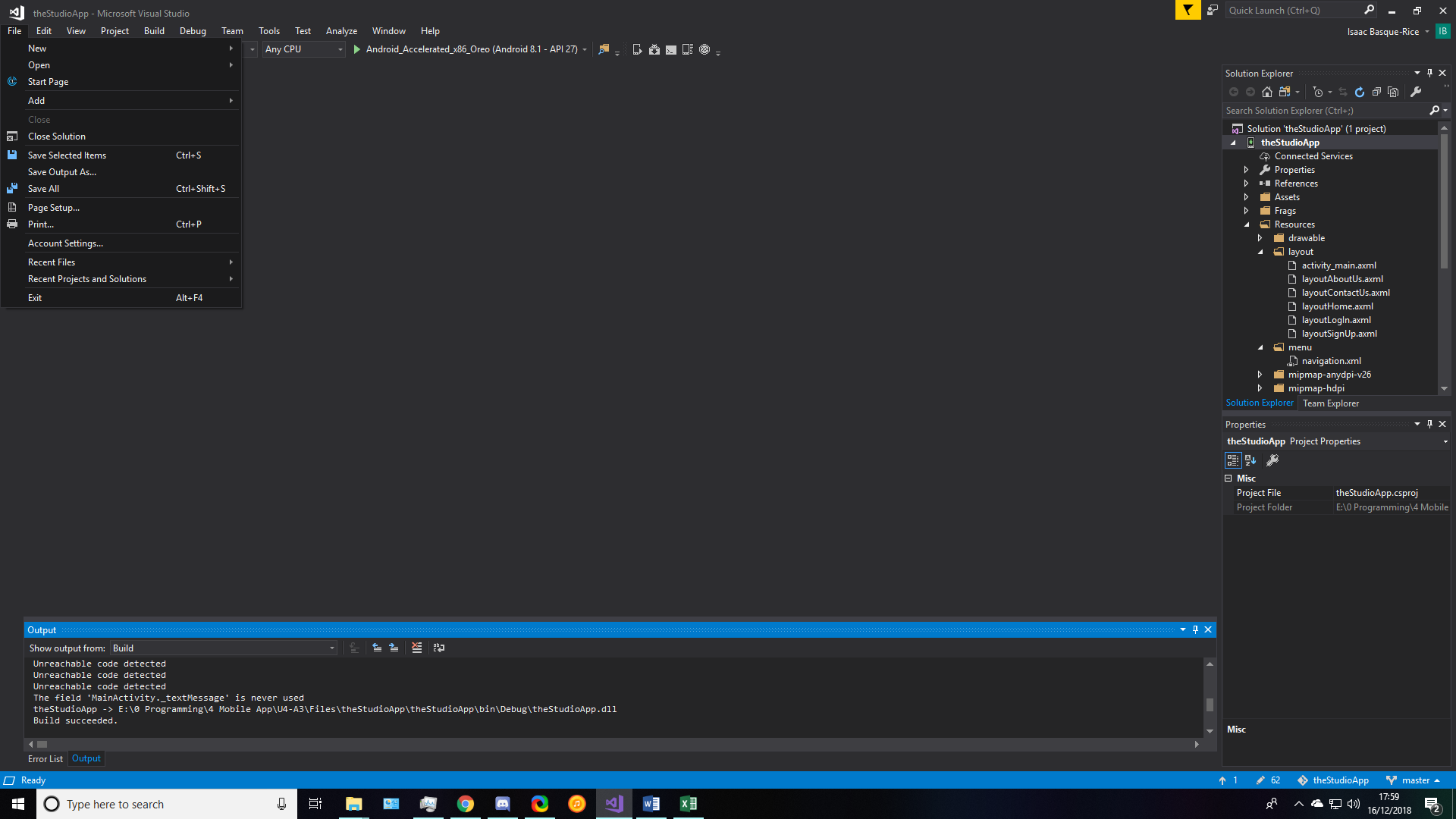


Screenshot 13, the undo and redo buttons

Saving can be done by pressing Ctrl+S, by pressing the various save buttons as seen below, or by going to File > Save/Save As/etc., also seen below.

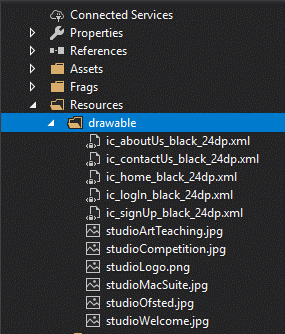


Screenshot 14, Save Selected Items and Save All, respectively



Screenshot 15, in the File menu

Other notable functionalities in Visual Studio are the ability to create new projects, done in various ways including through the start menu as displayed in screenshot 10 or through ctrl+shift+N, and the ability to add multimedia to almost any piece of development work through the solution explorer and a reference to the image’s location in code.

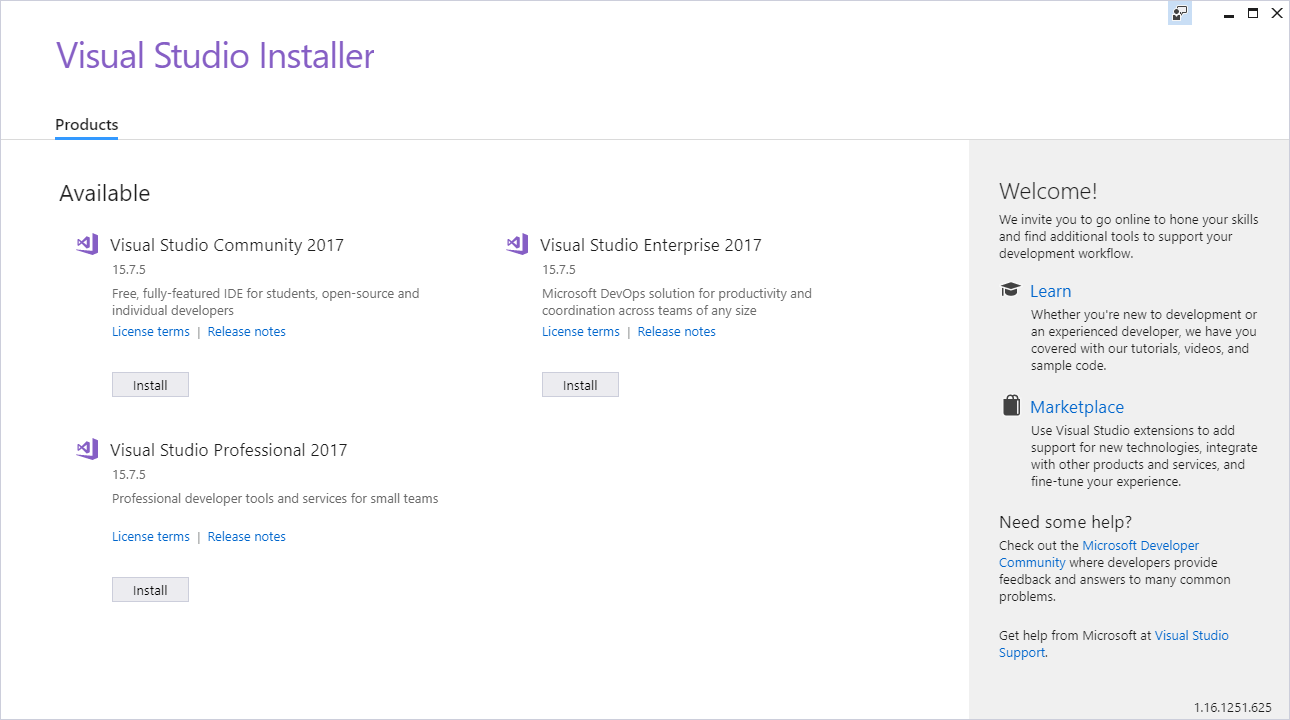


Screenshot 16, the assets, or 'drawable' folder in my app's solution.

Finally, just to touch on debugging once more, one of the core principles of any development environment is the ability to run the developer’s code, which (given that the developer doesn’t change or alter any settings) very rarely indeed presents a problem to the developer. In this instance, however, through the usage of the cross-compiler, running the program may present some challenges to me as due to the app’s status as a mobile application Visual Studio will first need to launch an Android emulator and then run the app on that, which will take extra time and effort to set up.

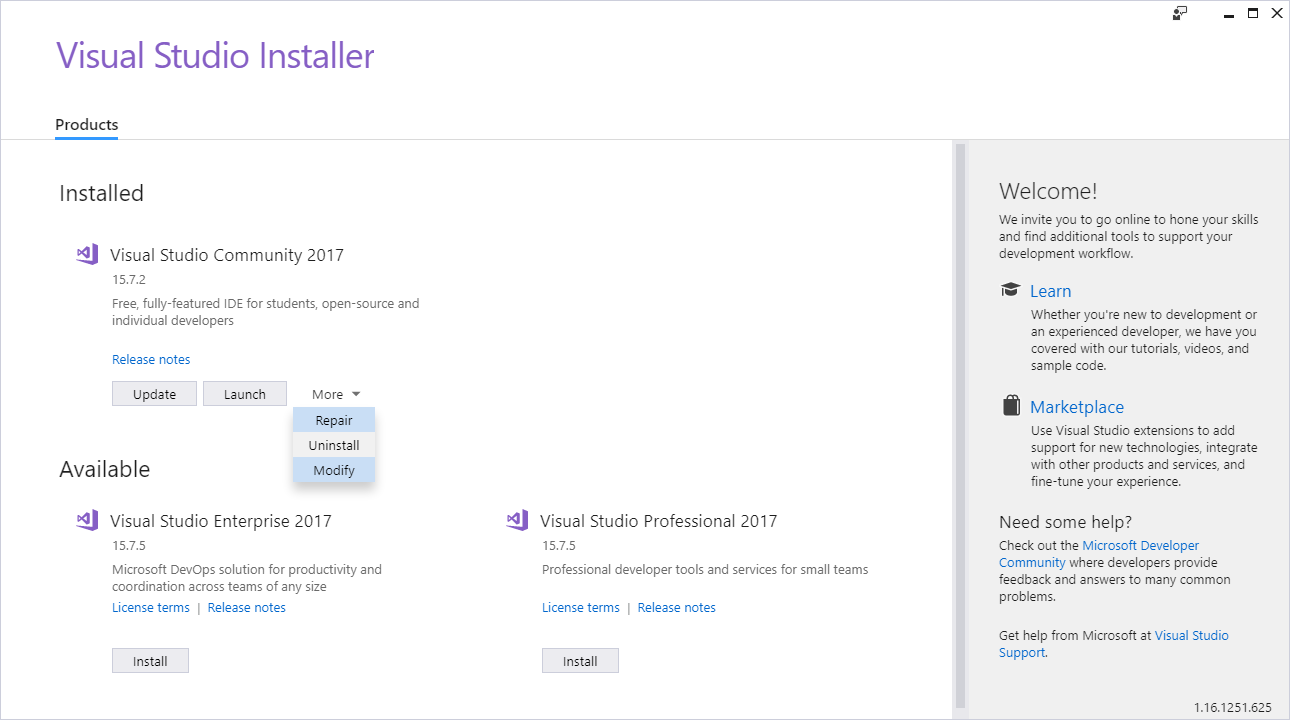
## Installation and Configuration

The installation for Visual Studio was fairly seamless, downloading the installer program and selecting specific packages to install into the IDE caused no issues whatsoever. Below is a screenshot of the screen that greets you upon launching the installer program, this allows the user to install a specific version of visual studio, in my case, Community:

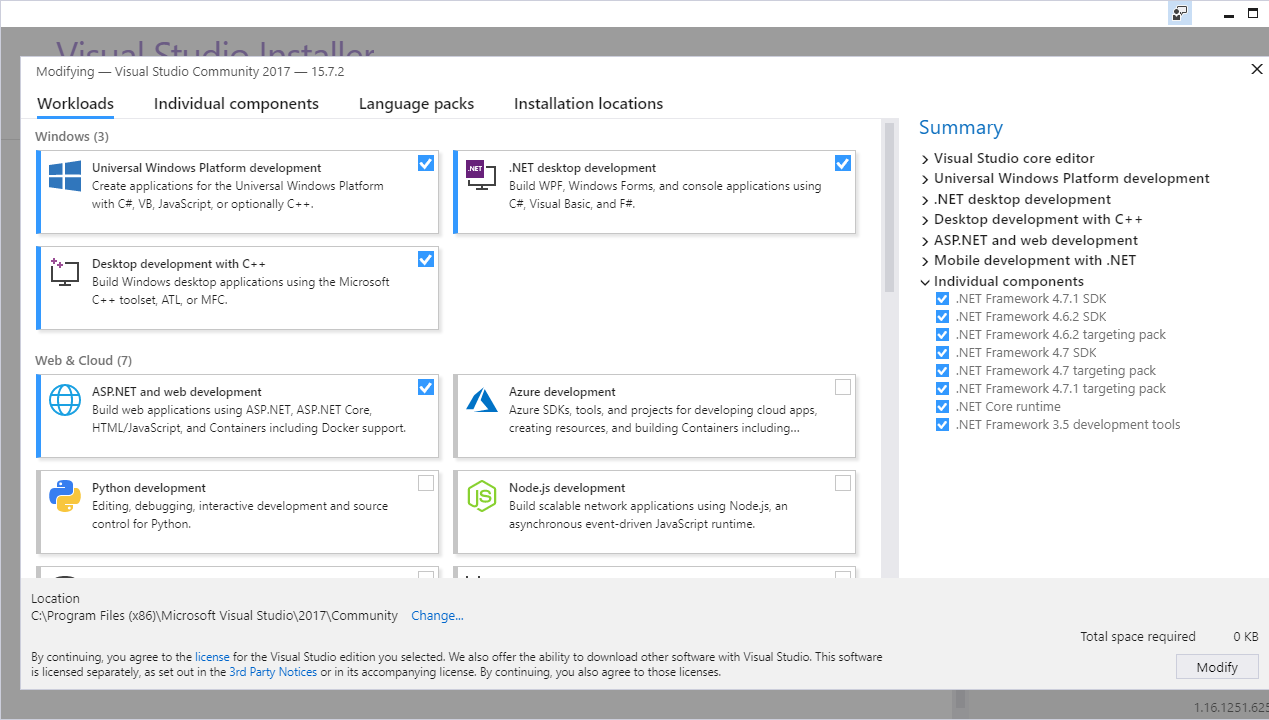


Screenshot 17, the landing screen for the VS installer, if Visual Studio is not yet installed

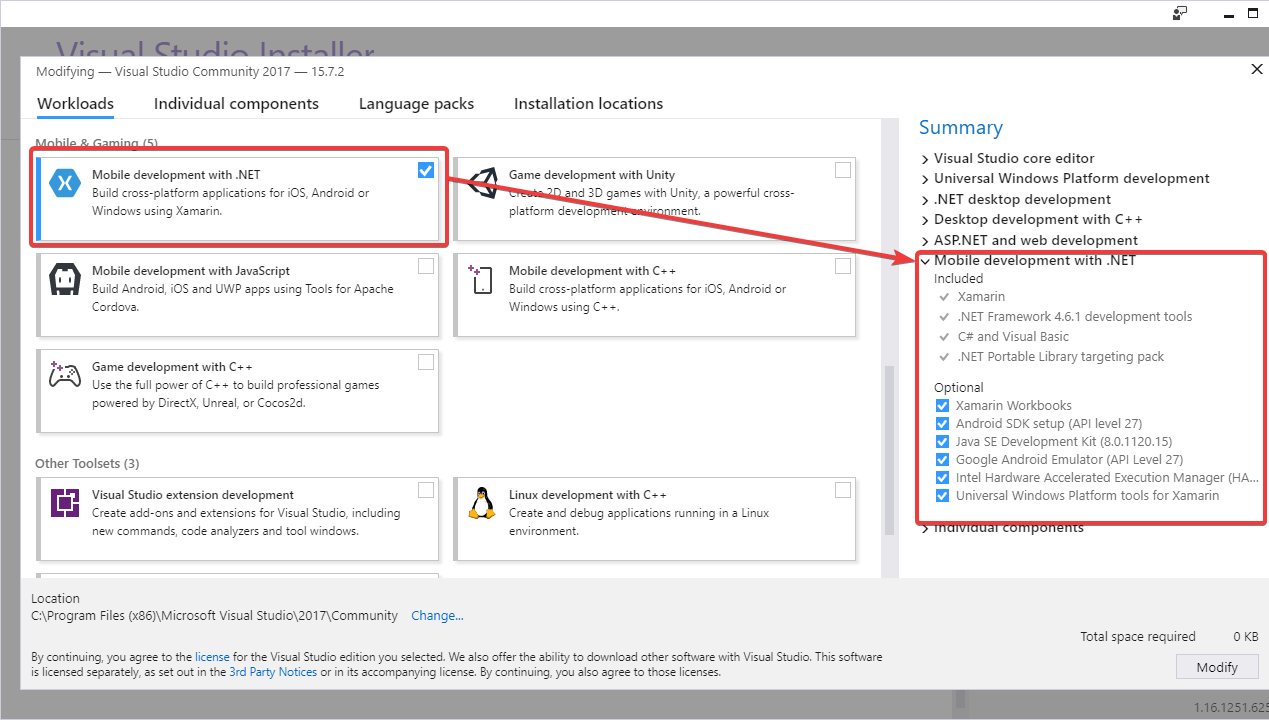
Upon installation I selected an option known as ‘modify’ from a drop-down menu which allowed me to select specific packages I wished to make use of in my capacity as a developer, these packages were the standard windows development packages plus ASP.NET web development and, most crucially in this context, ‘Mobile development with .NET’, the following is a series of screenshots demonstrating this:



Screenshot 18, the landing screen for the VS installer, if Visual Studio is installed but requires an update



Screenshot 19, a selection of packages you may add to Visual Studio



Screenshot 20, the Xamarin package, what I will be using for my app

Additionally, I decided that I needed to update Visual Studio at the start of this project, as the screenshot below describes:



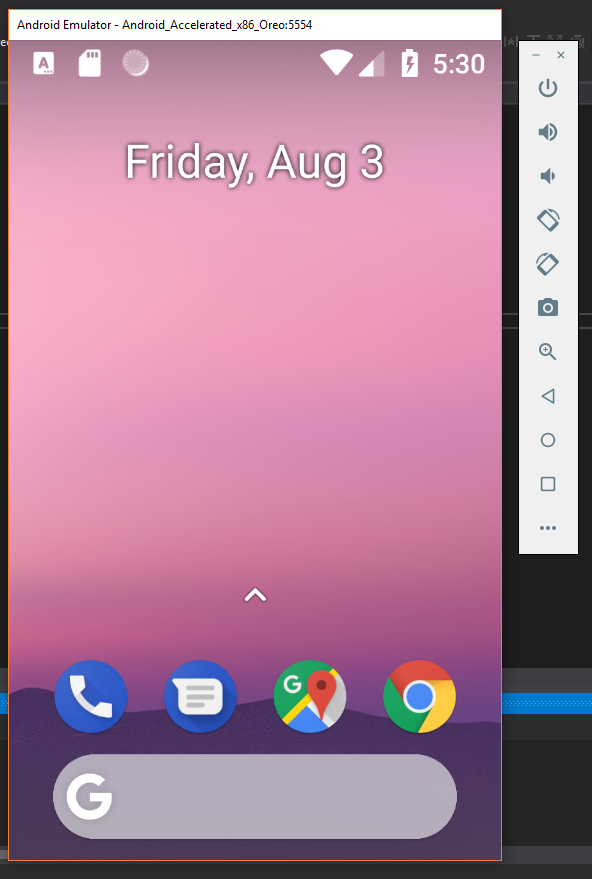
Screenshot 21, Visual Studio updaing

This will only be happening once as updating constantly may affect the development cycle and final project due to the possible addition or removal of specific aspects of the IDE I may or may not require for the project.

## Emulators and physical devices

In the process of creating my application I am required to make use of both an emulator and a physical hardware device in order to test and debug my application, both of these have positive aspects as well as setbacks, as such, this section has been created to serve as an explanation as to these pros and cons, if you will, as well as an explanation of the separate devices and their differences.

Firstly, an emulator allows for the debugging of my program in an environment that simulates (emulates) the environment that the application would be running on normally, in my case I will be using the inbuilt Android 8.1 emulator within Visual Studio. Below is a screenshot of the emulator I will be using upon start-up:



Screenshot 22, the emulator I will use

The buttons along the side will serve as hardware buttons of sorts, in order from top to bottom: Power, volume up, volume down, turn left, turn right, camera, zoom, go back, home, and menu, as well as a button that takes you to an emulator settings panel which allows one to tweak the finer details of the emulation process such as battery life, time displayed etc.

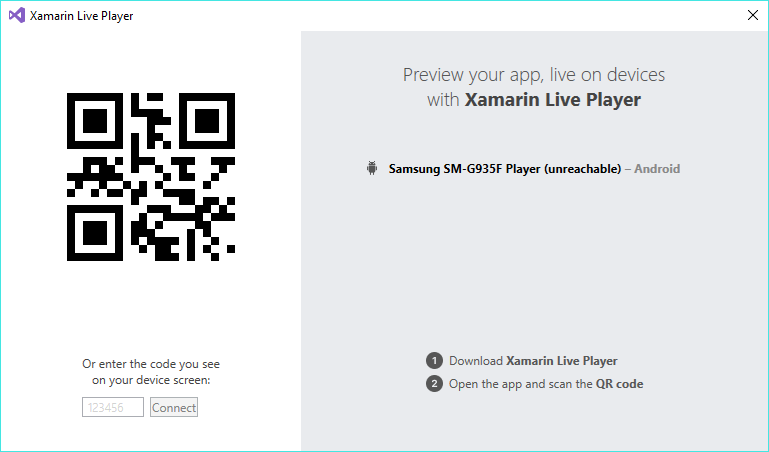
Next, we have a physical device that I will be making use of in the process of development, my own personal Samsung Galaxy S7 Edge, pictured below:



Photo 1, my phone

This phone has the Xamarin Live Player installed on it, which allows a phone on the same network as the development computer to display an application in development wirelessly on a phone. This will be the primary method of debugging for several reasons, firstly, for convenience’s sake, as well as for the sake of the user as the app won’t usually be run within an emulator but rather on an actual smartphone, as a result testing on an actual smart device seems to be the most sensible option.

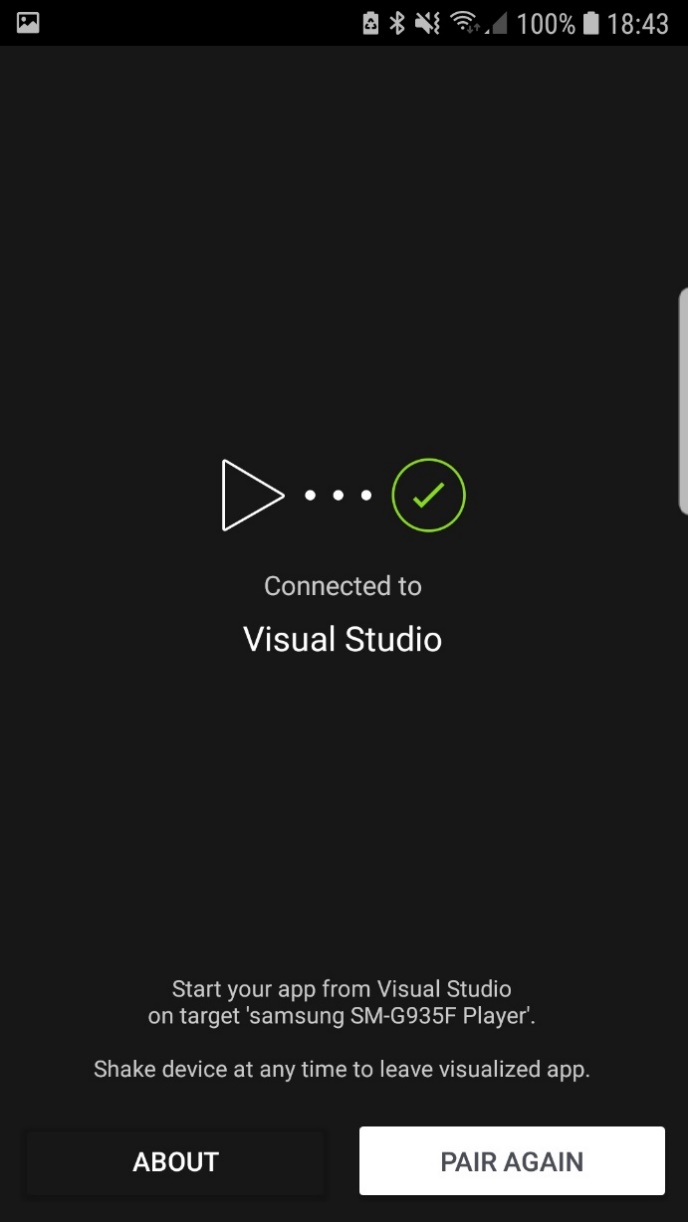
Below are some more screenshots, this time of the Xamarin live player, the first screenshot is of a QR code, of which my smartphone will scan, this allows the computer to confirm I am viewing it from my phone and launch the app. The second and third screenshots are from within the Xamarin live player app, the first of the two showing that it’s waiting for a QR code to scan, and the last being confirmation that the app is connected to visual studio



Screenshot 23, the Xamarin Live Player computer dialog window



Screenshot 24, the Live Player app as inatalled on my phone, waiting for a connection



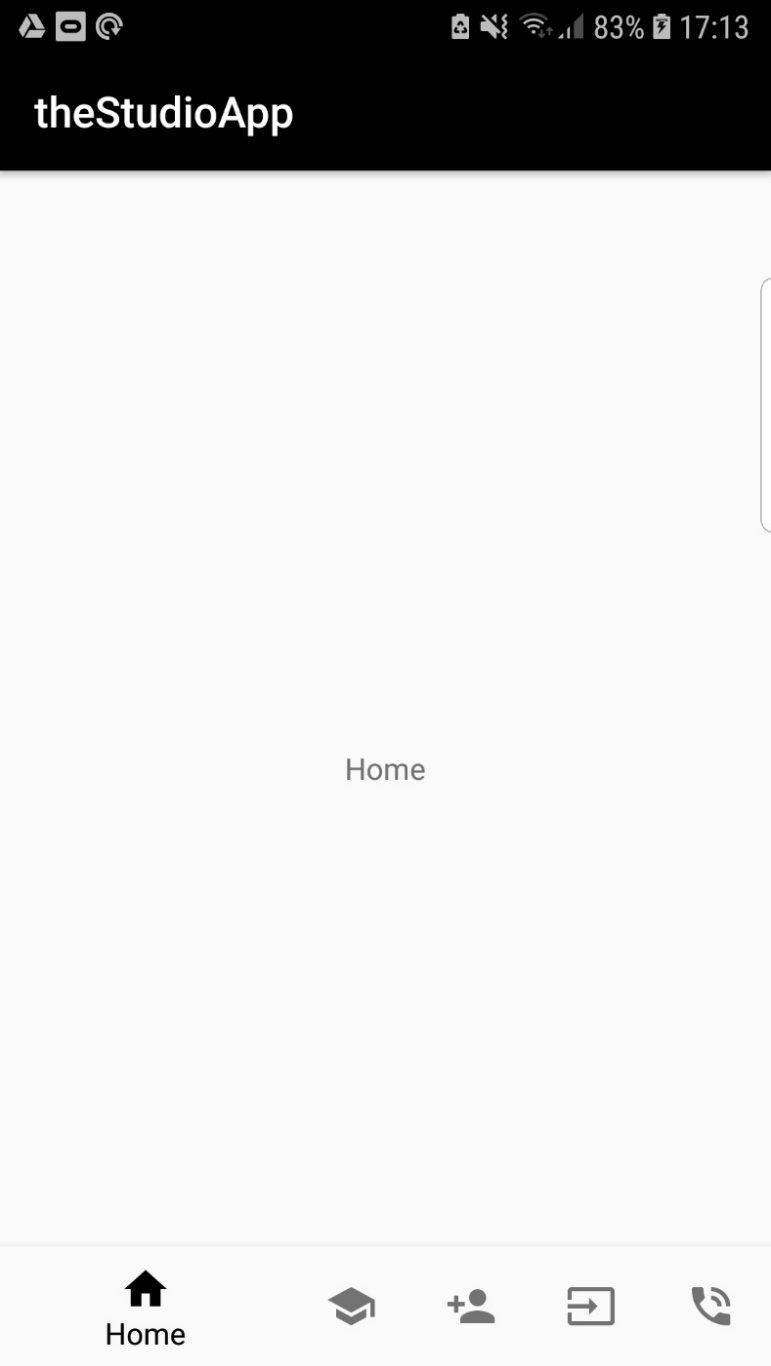
Screenshot 25, the live player app connected to VS

# Creation and deployment of a working mobile app

## Appropriate features and functions

Any and all functional mobile applications must have a range of specification appropriate features and functions. The specification for this application states that as well as static content displaying on the application’s screen a sign up and log in screen is required, this results in the requirement of a fully functional back end to my application, powered, in this instance by a SQLite database system. As such in this document I have decided to split the two into frontend and backend development tasks, in frontend I will be specifying all I am doing to keep the appearance of the application good, XML files, images, and so on and so forth, and in the backend section I will be detailing all that goes in to the backend of the app, as one could Imagine.

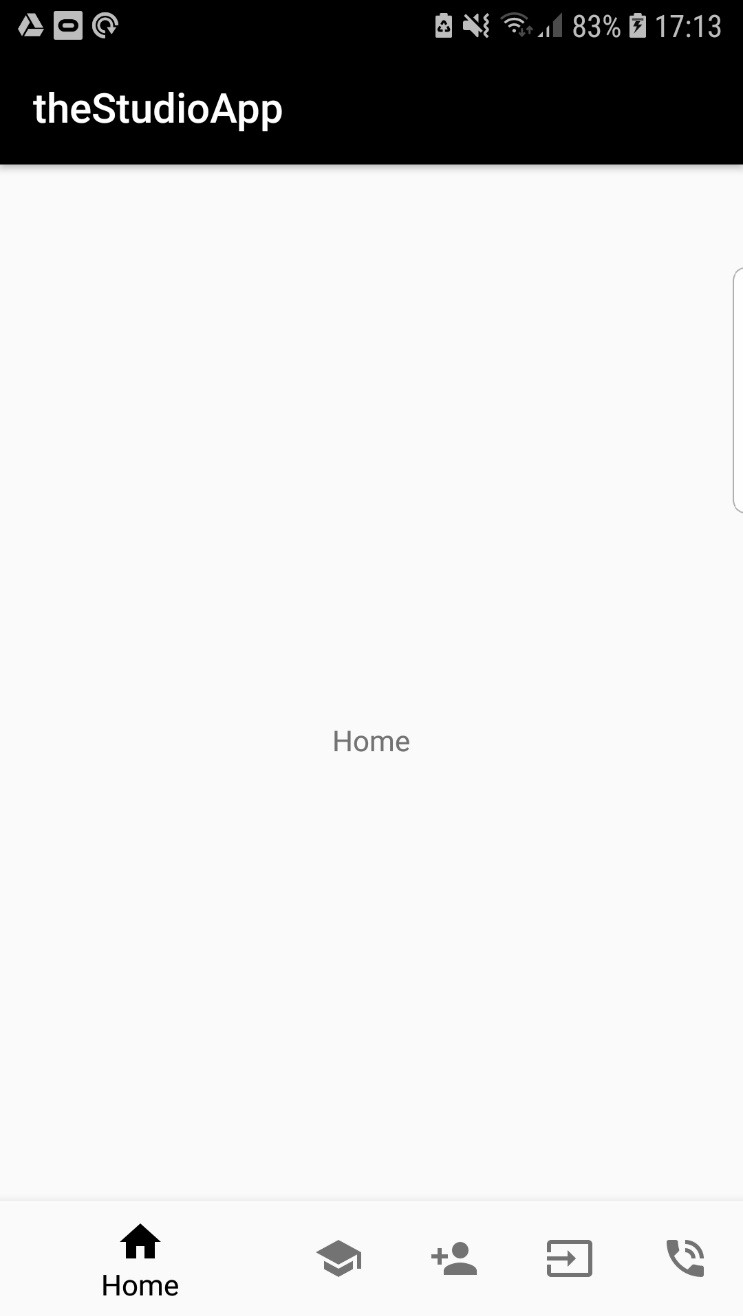
Below we see an on-device screenshot of the beginning stages of the application, this is mostly encapsulated by the addition of four separate screens that I will use for the basis of the structure of the application. These screens are Home, About Us, Sign Up, Log In, and Contact Us, each described by their own material design icon (found here: <https://material.io/tools/icons/?style=baseline>). Additionally, the banner along the top has been changed to #000000, or true black, in order to conform to the client’s design requirements. Finally, on each screen in the centre we have the title of the screen as a place holder of sorts, below we see one example that of the ‘home’ screen.



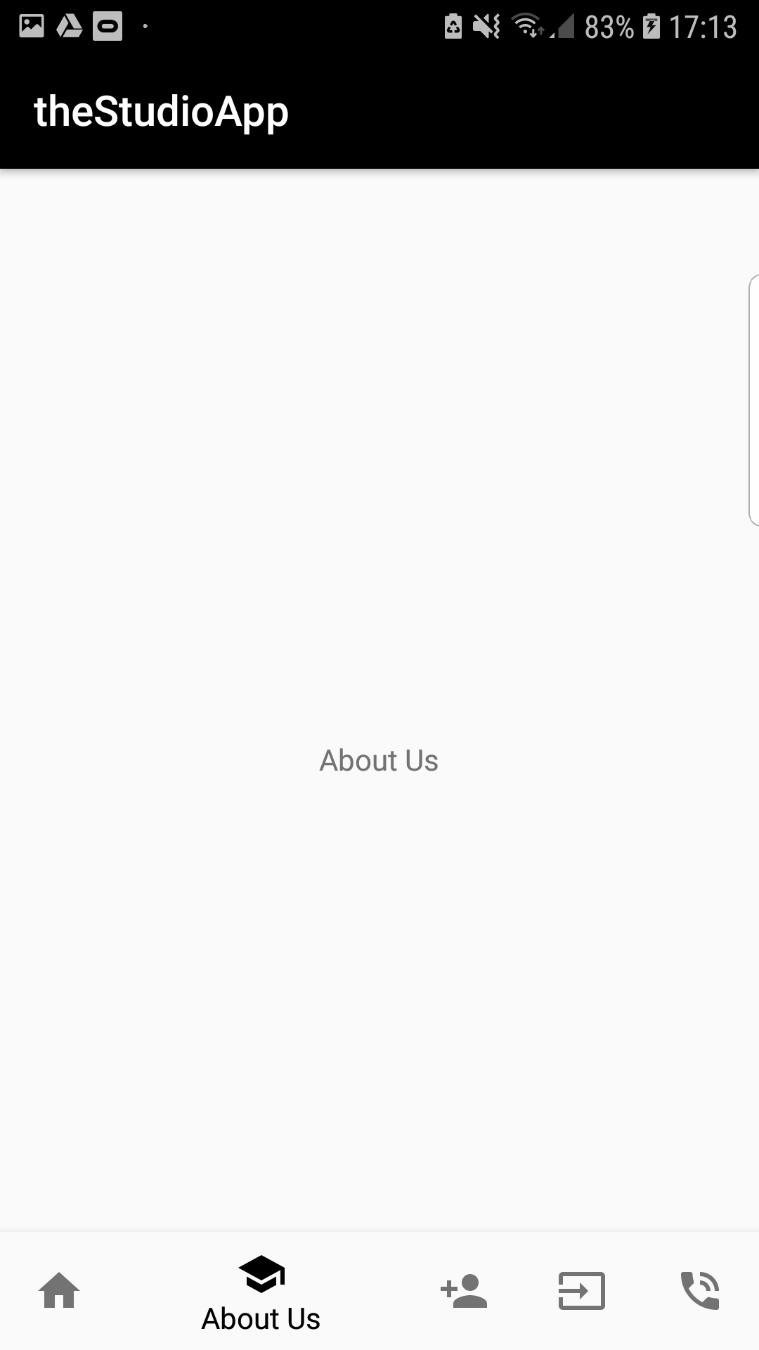
Screenshot 26, my app without content but with nav bar and colouring

### Navigation

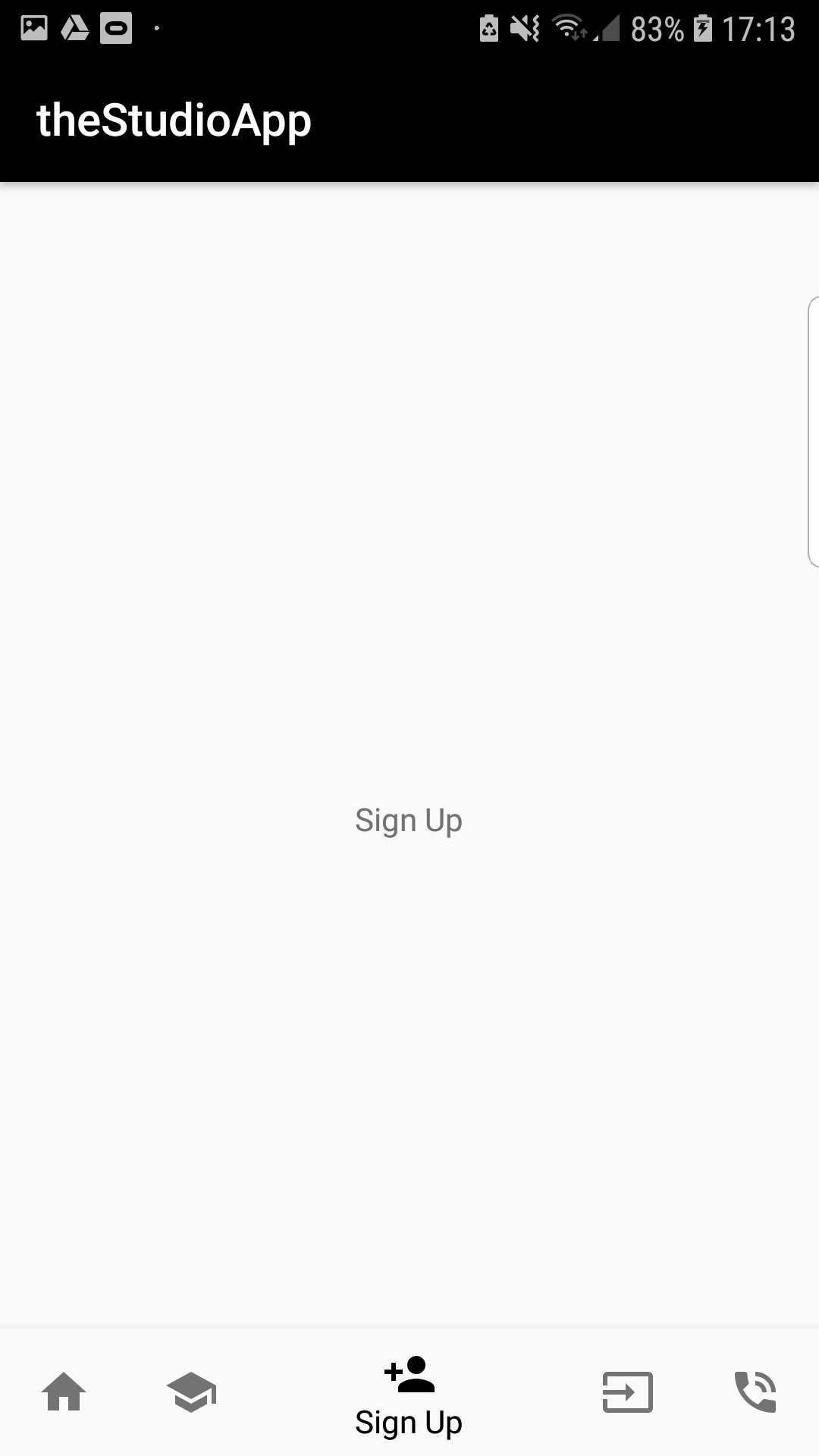
Below now you will see the bottom navigation bar, due to width constraints on most mobile devices (for example the development device, a *Samsung Galaxy S7 Edge*), and also due to the number of screens one may require for this application, the design of this section has naturally taken on an accordion-like quality, displaying only the title of the screen the user is currently on, an ideal next step would be to incorporate the client’s proprietary yellow colour (#f7d31d) into this



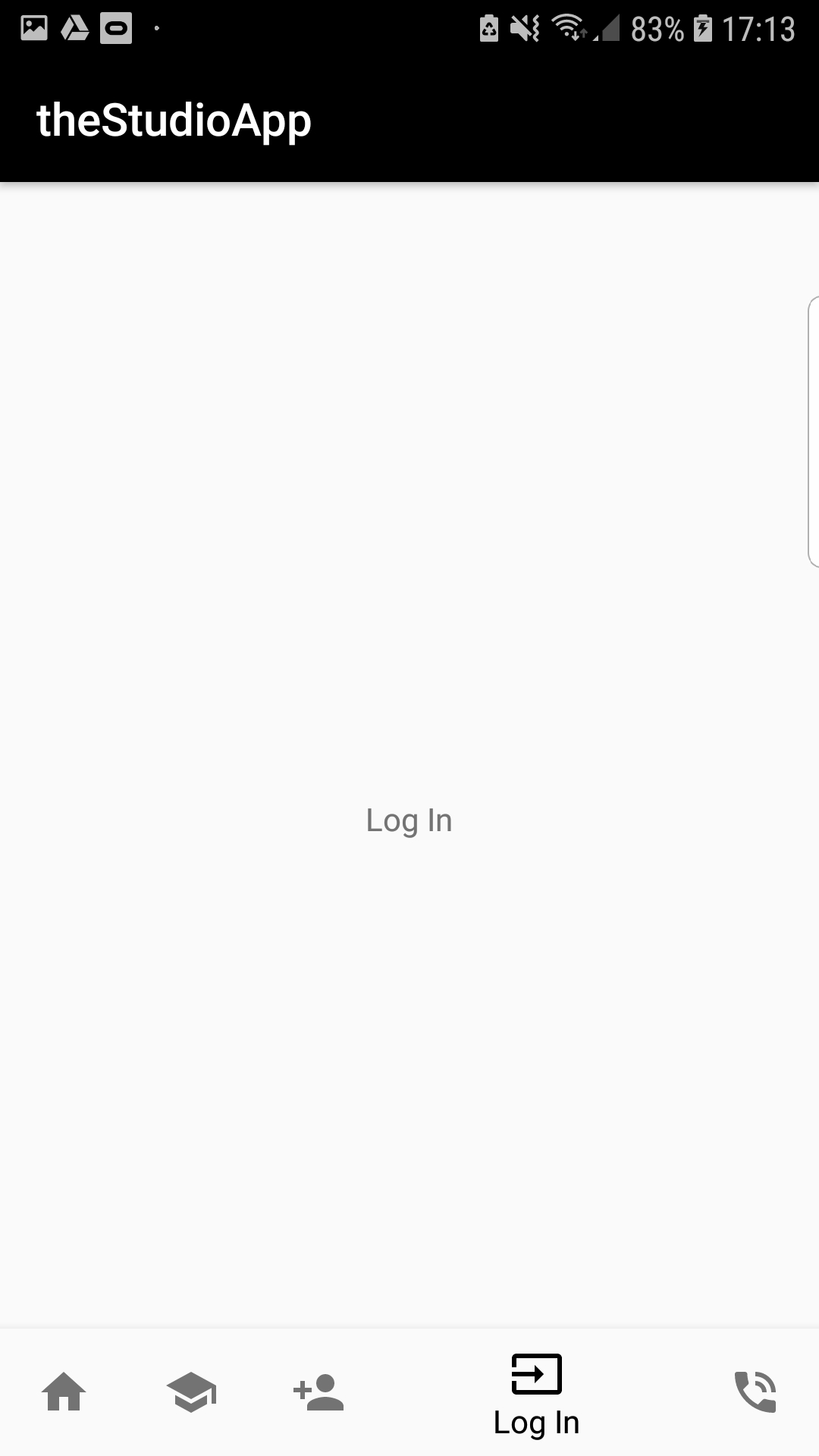
Screenshot 27, The appearance of the nav bar when the Home tab is opened



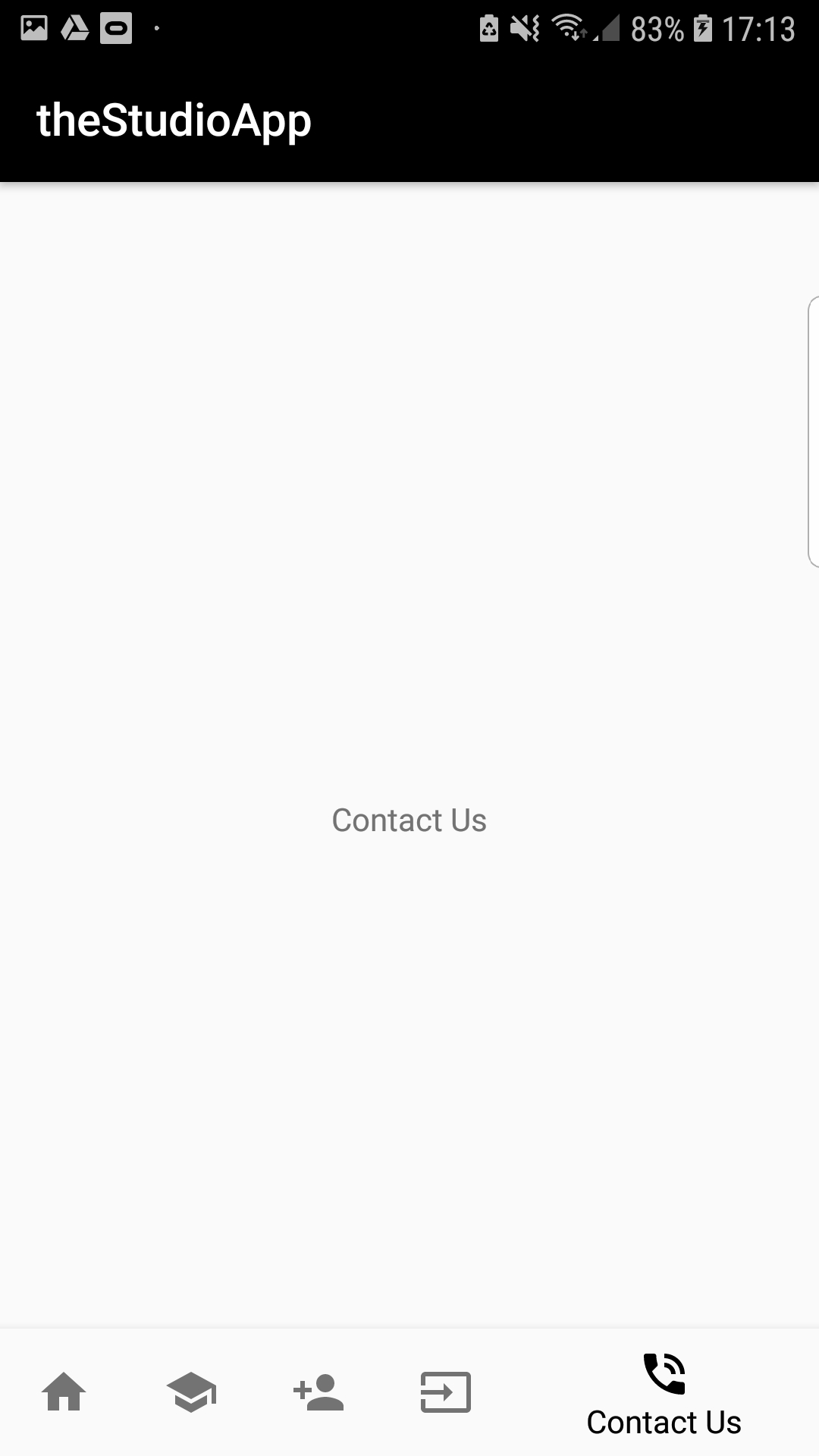
Screenshot 28, The appearance of the nav bar when the About Us tab is opened



Screenshot 29, The appearance of the nav bar when the Sign Up tab is opened

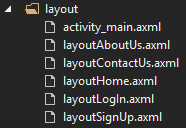


Screenshot 30, The appearance of the nav bar when the Log In tab is opened

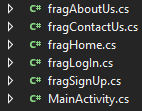


Screenshot 31, The appearance of the nav bar when the Contact Us tab is opened

Next, in order for the different screens to display separate things (Text on one screen and forms on another, as an example), I had to make use of so-called fragments within my code. These fragments will represent individual screens and be made up of two separate files, the XML files that will be responsible for the physical appearance and styling of the application, and the C# files that represent the code behind, such as the SQLite database etc., below are the file structures of each of these, note that both have, in addition to the files corresponding to the individual screens, a ‘main’ file, this is for basic information that will exist within the context of the entire app (For example, the navigation bar, supported by an additional xml file connected to activity\_main).



Screenshot 32, the Layout section of the solution files



Screenshot 33, the C# fragments th the solution

The following is the code that allows for fragmentation, annotated with language and file location:

XML located in activity\_main.axml

<!-- basic formatting for fragments -->

<FrameLayout

android:layout\_width="fill\_parent"

android:layout\_height="fill\_parent"

android:layout\_marginLeft="8dp"

android:layout\_marginRight="8dp"

android:id="@+id/fragMainContainer"

android:tag="MainFrag"

android:layout\_above="@id/navigation"

android:layout\_below="@id/navigation" />

C# Boolean (true/false variable) located in MainActivity.cs

//if a user navigates to screen X create fragX and switch to it

case Resource.Id.navigation\_home:

SwitchToFrag(new fragHome());

return true;

case Resource.Id.navigation\_aboutUs:

SwitchToFrag(new fragAboutUs());

return true;

case Resource.Id.navigation\_signUp:

SwitchToFrag(new fragSignUp());

return true;

case Resource.Id.navigation\_logIn:

SwitchToFrag(new fragLogIn());

return true;

case Resource.Id.navigation\_contactUs:

SwitchToFrag(new fragContactUs());

return true;

C# class located in MainActivity.cs

//class that allows switching between fragments

protected void SwitchToFrag(Fragment frag)

{

FragmentTransaction transaction = FragmentManager.BeginTransaction();

transaction.Replace(Resource.Id.fragMainContainer, frag).SetTransition(FragmentTransit.FragmentFade).Commit();

}

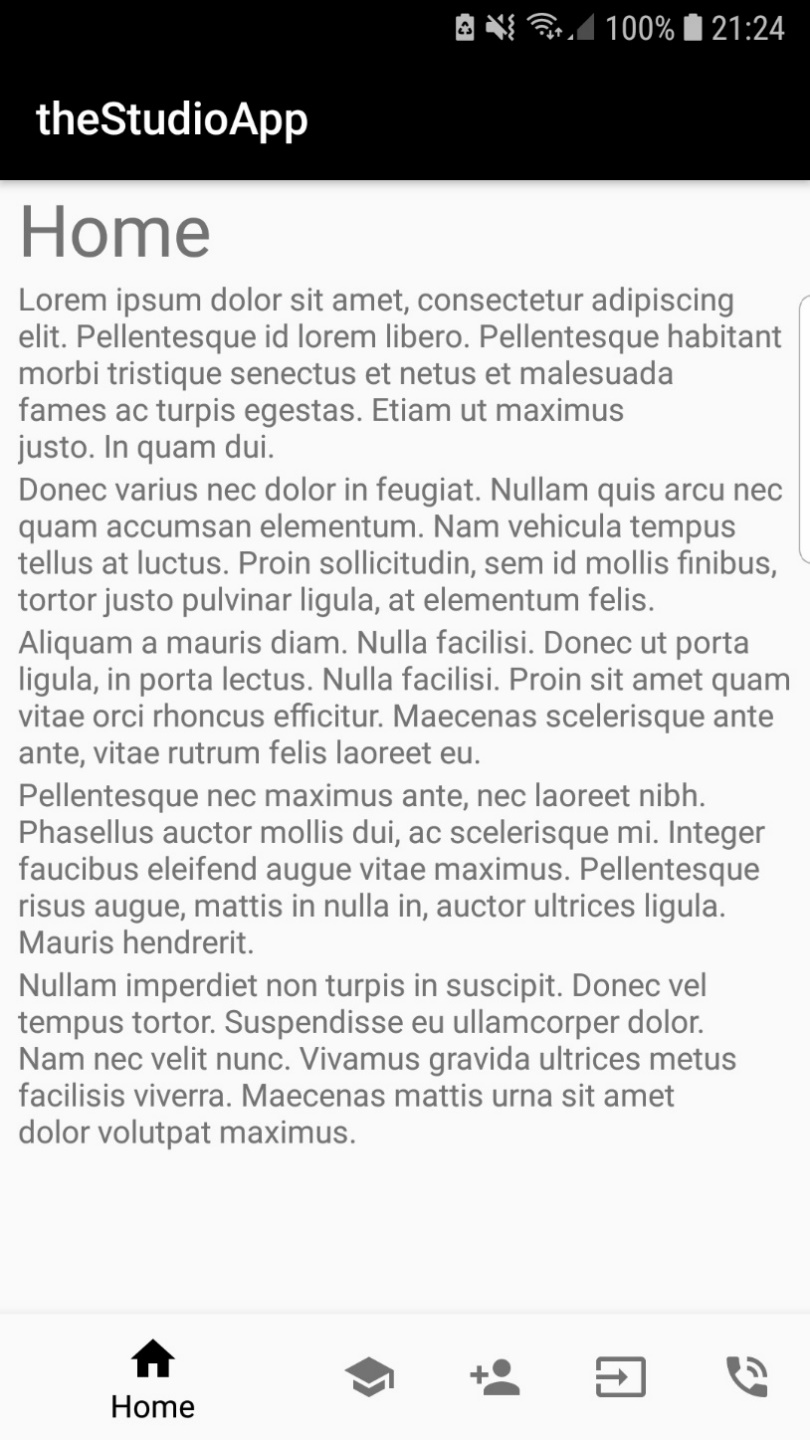
In addition to all this it has become clear that the app will require a description for use on the store. The description must have no more than 255 characters, be from the perspective of the user, and highlight some features of the app, as such, what follows is the official description I will use.

The Studio, Liverpool is a new and innovative school for 14-19-year olds. Now with theStudioSchool app you able to keep up to date with the happenings of the school AND contact the school with ease, the must have app for all studio students and parents!

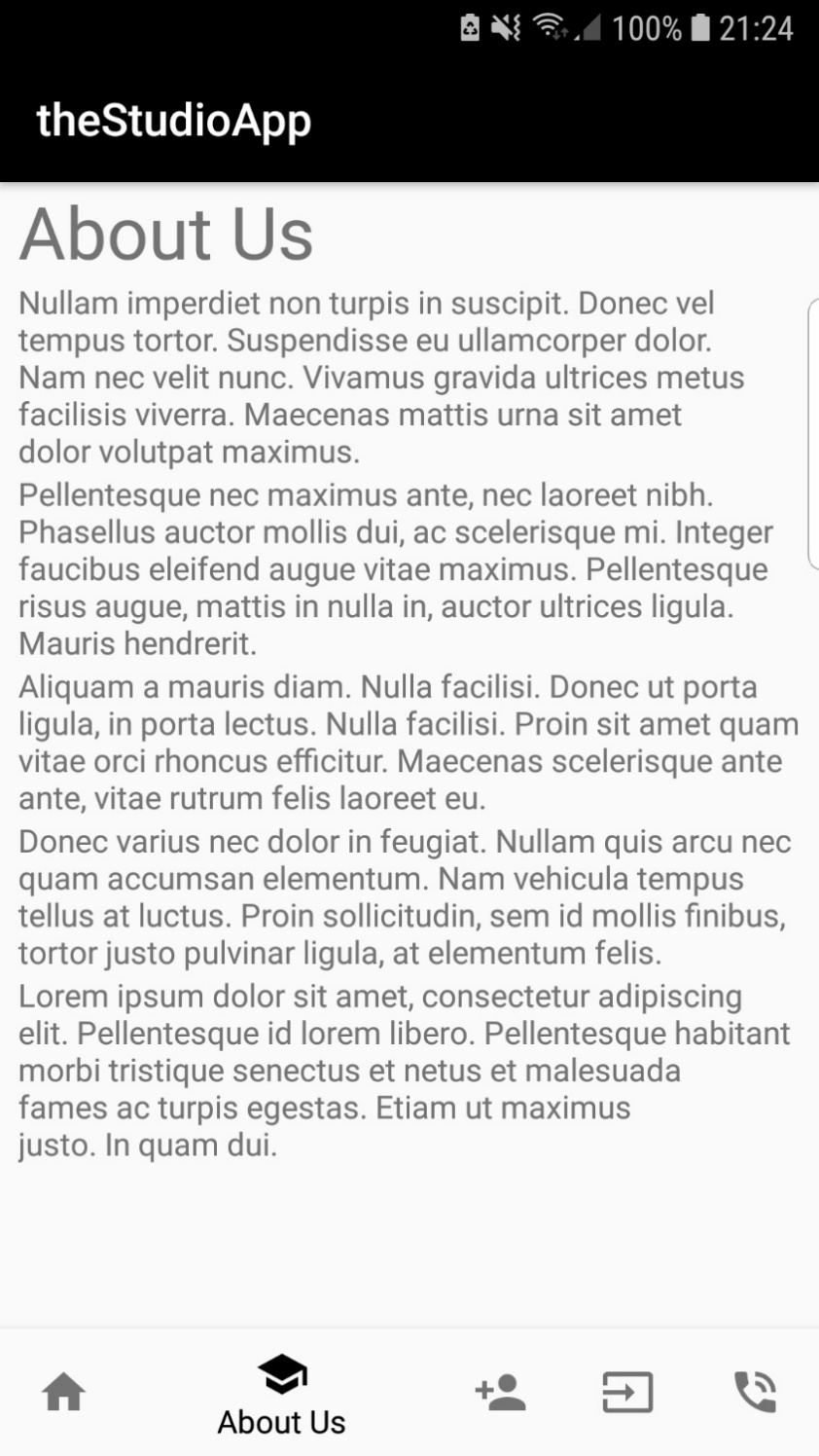
### Content

What follows is the application in a vaguely more appealing state. The “Home” and “About Us” screens contain the same five paragraphs of 32 words of Lorem Ipsum text rearranged to show the textual capabilities of the application. In addition to this, each of the five screens has a title corresponding to the name of the screen, this text is not different to any of the other text apart from the fact that it is in 32dp font size as opposed to the standard size.

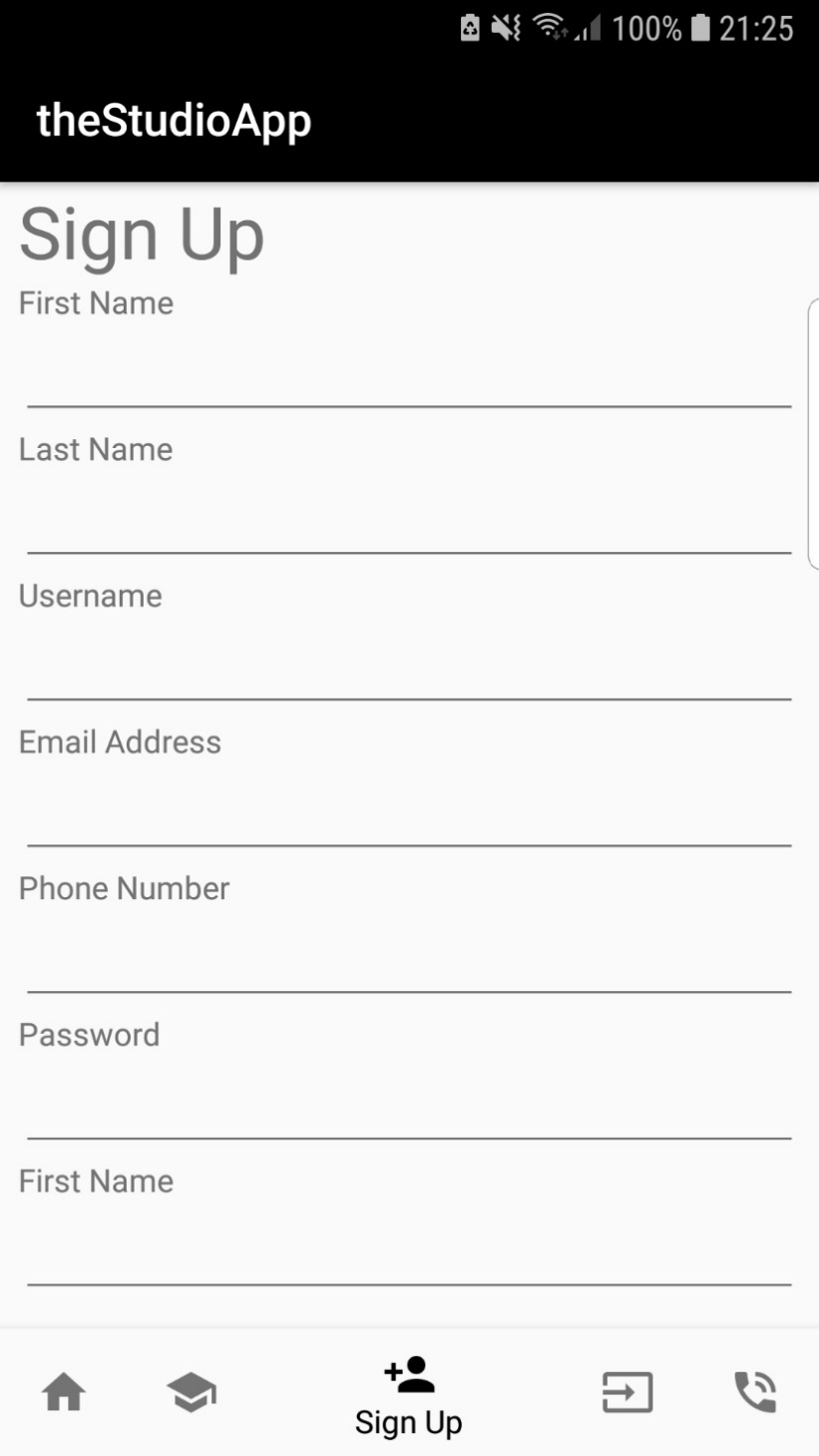
Additionally, in the latter three screenshots on display we see a series of text input fields corresponding to a user’s first name, surname, email address etc. In future development updates I will be adding SQLite functionality to the backend of the application allowing the signing in and registering of users using this method of data input.



Screenshot 34, the home tab with a selection of Lorem Ipsum paragraphs

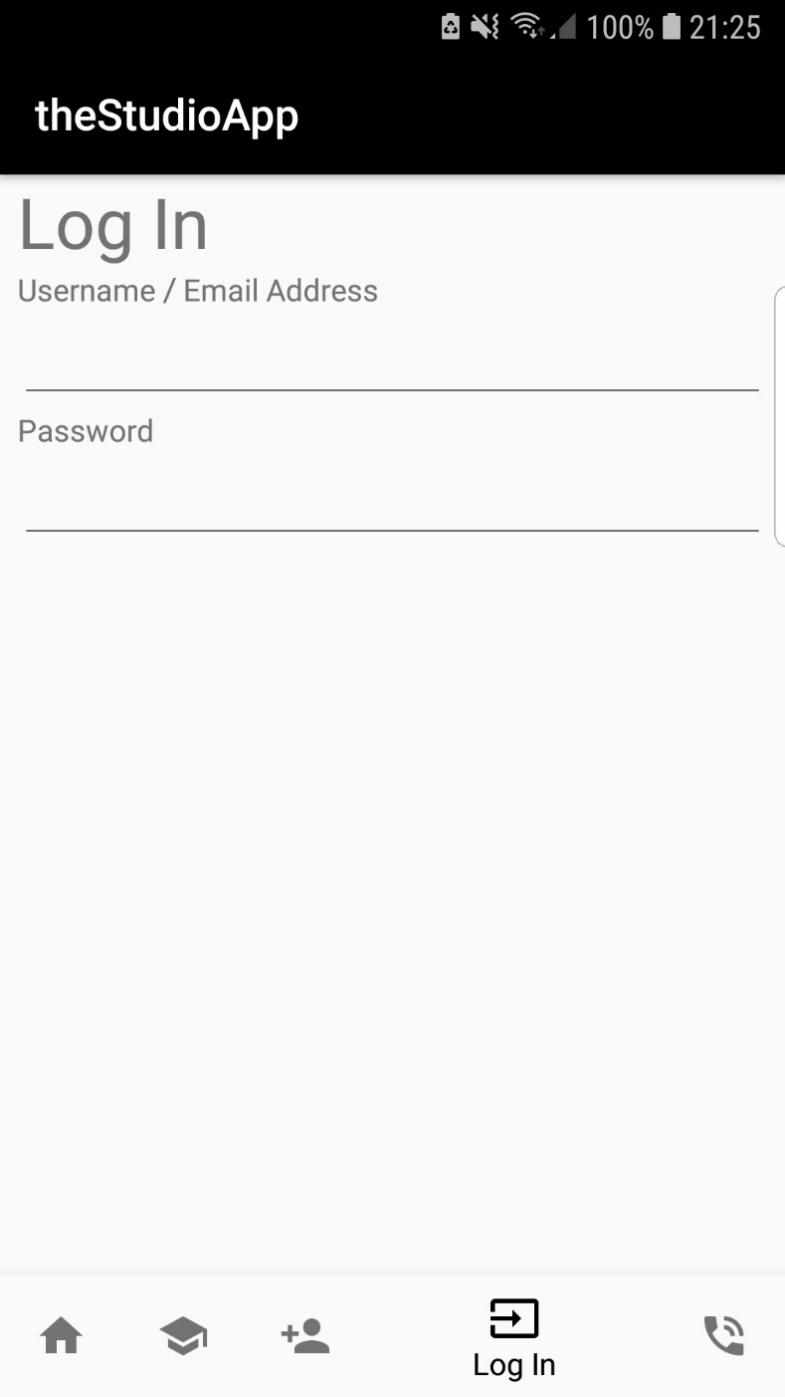


Screenshot 35, the about us tab with a selection of Lorem Ipsum paragraphs



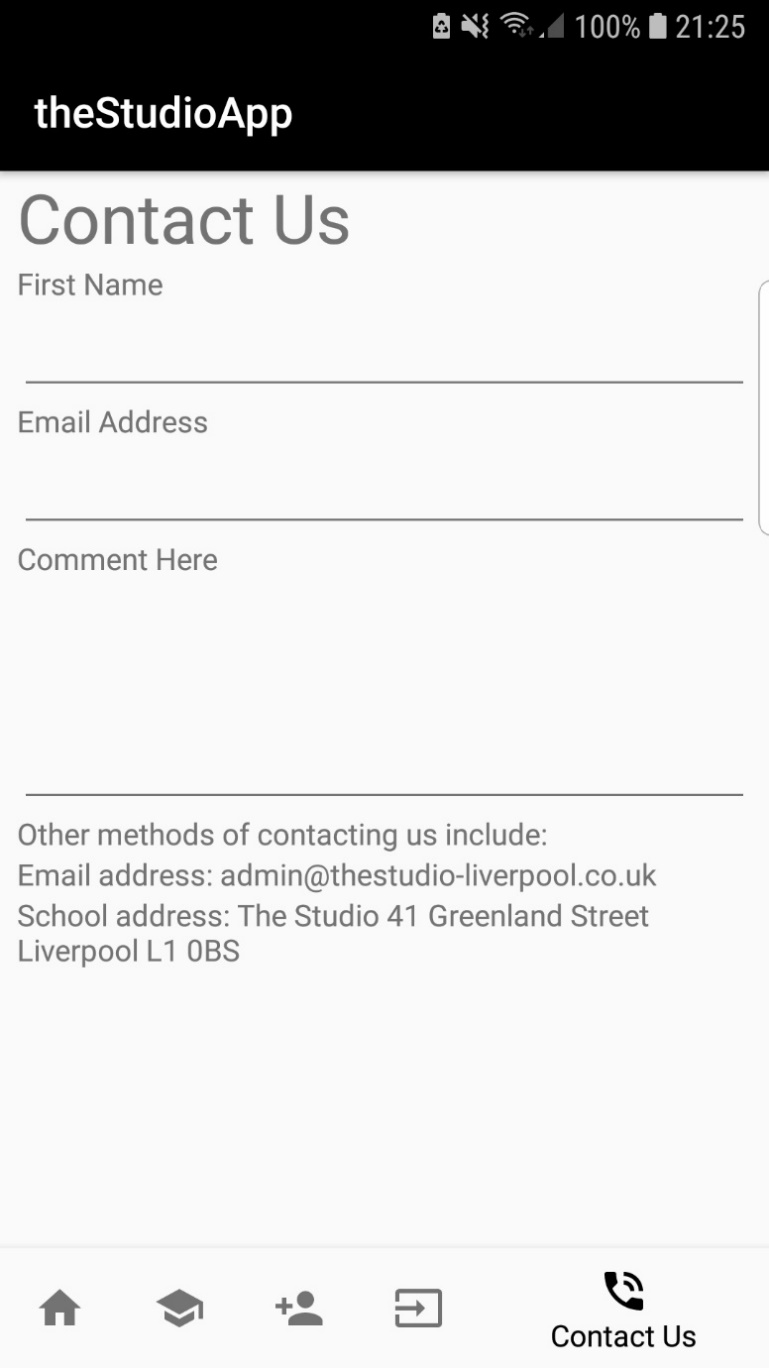
Screenshot 36, the Sign Up tab with near fully featured front end (forms)[[1]](#footnote-1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Control | Properties | Name | Handlers | Purpose |
| Button | Font: Roboto  Text Size: 42  ForeColour: #f7d31d  BackColour: #000000 | btn\_Nav (when referred to as a group of 5) | OnClick | Switch from screen to screen between my application |
| Textbox | Font: Roboto  TextSize: 48  Colour: #000000 | txtFirstName,  txtLastName,  txtUserName, | N/A | Allows user to input their first name, surname, and username respectively |
| Textbox | Font: Roboto  TextSize: 48  Colour: #000000 | txtEmailAddress | N/A | Allows user to input email, has an email validation regex statement:  “^\w+([-+.']\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*$” |
| Textbox | Font: Roboto  TextSize: 48  Colour: #000000 | TxtPhoneNumber | N/A | Only accepts int inputs |
| Textbox | Font: Roboto  TextSize: 48  Colour: #000000 TextReplace: Password | txtPassword | N/A | Replaces all character with a dot character, allows users to input passwords |
| Button | Font: Roboto  TextSize: 42  ForeColour: #000000  BackColour: #f7d31d | btnSubmit | OnClick | Allows users to enter their personal information into the database |



Screenshot 37, the Log In tab with near fully featured front end (form)[[2]](#footnote-2)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Control | Properties | Name | Handlers | Purpose |
| Button | Font: Roboto  Text Size: 42  ForeColour: #f7d31d  BackColour: #000000 | btn\_Nav (when referred to as a group of 5) | OnClick | Switch from screen to screen between my application |
| Textbox | Font: Roboto  TextSize: 48  Colour: #000000 | txtUserNameAndEmail, | N/A | Allows user to input their Username OR Email address |
| Button | Font: Roboto  TextSize: 42  ForeColour: #000000  BackColour: #f7d31d | btnSubmit | OnClick | Allows users to check their information against the database and sign in |



Screenshot 38, the contact us screen with near fully featured front end (form)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Control | Properties | Name | Handlers | Purpose |
| Button | Font: Roboto  Text Size: 42  ForeColour: #f7d31d  BackColour: #000000 | btn\_Nav (when referred to as a group of 5) | OnClick | Switch from screen to screen between my application |
| Textbox | Font: Roboto  TextSize: 48  Colour: #000000 | txtUserNameAndEmail, | N/A | Allows user to input their Username OR Email address |
| Button | Font: Roboto  TextSize: 42  ForeColour: #000000  BackColour: #f7d31d | btnSubmit | OnClick | Allows users to send their comment with first name and email address to a database, to be checked later |

The way that text in Android development works is that instead of directly inputting text, as one might in web development and in HTML, all text is input into a separate ‘strings.xml’ file and then referenced in the XML for the screen itself, the following code is an example of this:

<!--the first line declares the id of the text field, next two are formatting, and the final one references the text itself within strings.xml-->

<TextView

android:id="@+id/firstName"

android:layout\_height="wrap\_content"

android:layout\_width="wrap\_content"

android:text="@string/firstName" />

In strings.xml:

<string name="firstName">First Name</string>

And so, the words ‘First name’ will appear in the field, as in the screenshot above.

My next steps will be to add images, at this point some slight formatting issues may need to be fixed to make the front-end experience as smooth as possible, after this I will begin work on the back end, such as event handlers (buttons), and inputting data into an SQLite database.

In the backend the main feature is obvious, the database, the model of the database that I will be using is as follows, it contains, as mentioned previously, an ID, and one field for every editable text box in the solution:

// declares datatypes and fields

public class SignUpTable

{

public string firstName { get; set; }

public string lastName { get; set; }

public string username { get; set; }

public string emailAddress { get; set; }

public int phoneNumber { get; set; }

public string password { get; set; }

}

### Standards and Conventions

The C# coding conventions as laid out by Microsoft themselves (Wagner, et al., 2015) lays out in some detail the naming, layout, and commenting conventions that a developer would ideally stick to during development in order for their code to be universally readable. These conventions apply essentially entirely to the cases where a single developer is providing sole input into a program, e.g. where variables have to be named and so on. The article states the following:

Coding conventions serve the following purposes:

They create a consistent look to the code, so that readers can focus on content, not layout.

They enable readers to understand the code more quickly by making assumptions based on previous experience.

They facilitate copying, changing, and maintaining the code.

They demonstrate C# best practices.

Basic C# conventions state that layout should set indentation to four-character widths (when tab is pushed the curser moves along the width of four characters), one statement/declaration per line, and so on. Commenting conventions state that basic English grammatical rules apply, start with a capital letter, end with a full stop, and ensure every comment is on a separate line to the code it’s commenting, ideally above, as opposed to on the same line.

The final crucial part of C# conventions is naming conventions, and core to this are the concepts of camel casing and pascal casing. Due to the fact that it is not best practice to have spaces in variable names it naturally follows that all multi-word variables must be distinguished, this is where camelCasing and PascalCasing come in. in camelCasing the first word begins with a lower-case letter and all subsequent words with upper case letters to distinguish, however in addition to this PascalCasing has every individual word begin with an uppercase letter.

The difference between the two is essential to understand, PascalCasing is used for classes, constructors, methods, etc. however camelCasing is used for method arguments, local variables, field names, etc. in short, things that are camelCased inherit directly from things that are PascalCased. This distinction becomes important to understand when variables have one word in them.

The following is an example of code where camelCasing and PascalCasing are used in a crucial manner, due to the reasons above:

var localDistributors =

from customer in customers

join distributor in distributors on customer.City equals distributor.City

select new { Customer = customer, Distributor = distributor };

And now with the Pascal and camel sections highlighted (Key: PascalCased, camelCased):

var localDistributors =

from customer in customers

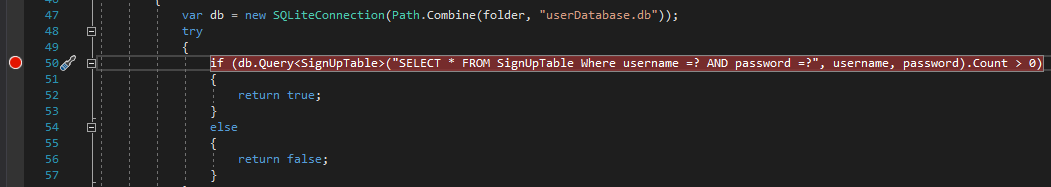
join distributor in distributors on customer.City equals distributor.City

select new { Customer = customer, Distributor = distributor };

Note that due to the fact that two local variables share names with two other classes, Pascal and camel were of the utmost importance in this instance.

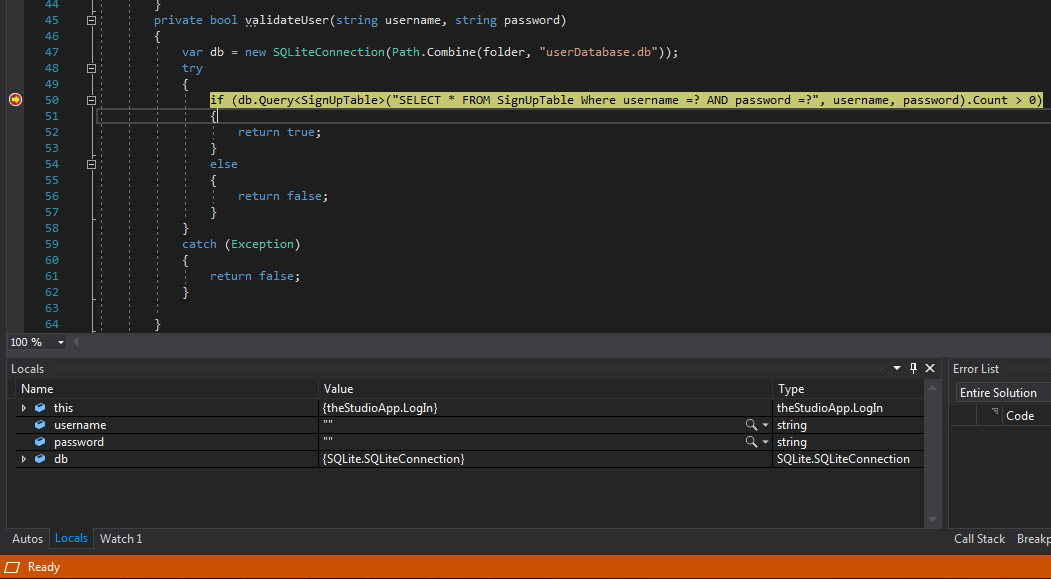
## Debugging tools and their uses

During development of this application there were numerous points at which bugs and issues in code appeared, this is apparent to any person who builds or maintains code and is a part of daily life in a development scenario. During the debugging process the usage of breakpoints is imperative as unhandled exceptions became an increasingly significant aspect of development with the introduction of the backend.



Screenshot 39, an example of a breakpoint in the backend of my code, it serves the log in page

Below is an example of a breakpoint in functioning code, the code’s purpose is to check, firstly, if anything is in fact in the log in fields, then it checks the details against data in the database, as one can see from the locals dialog nothing was inputted into the two text boxes and so the method returns false, this confirms that it is working as intended.

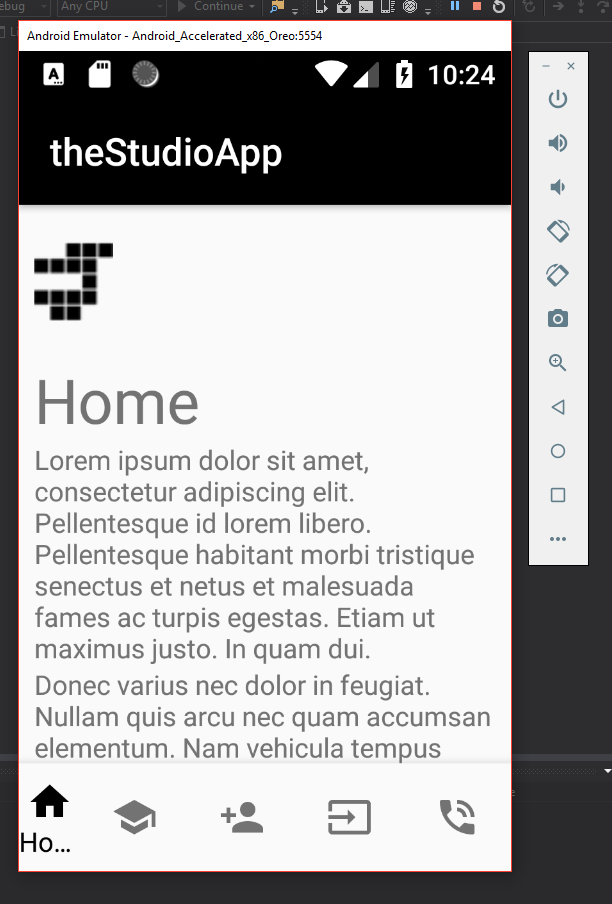


Screenshot 40, the breakpoint being hit

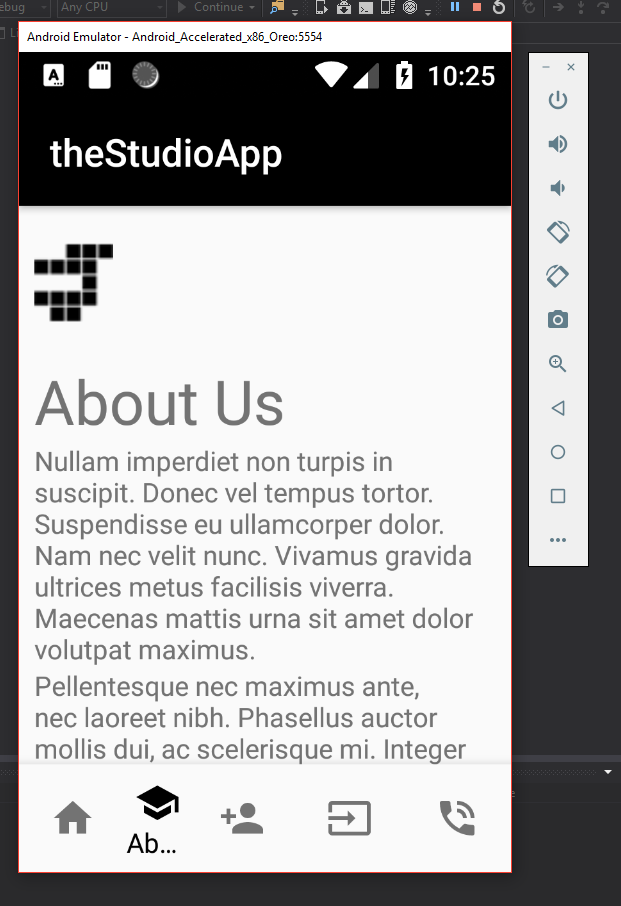
## Deployment to Virtual Device and Hardware Device

### Virtual Device

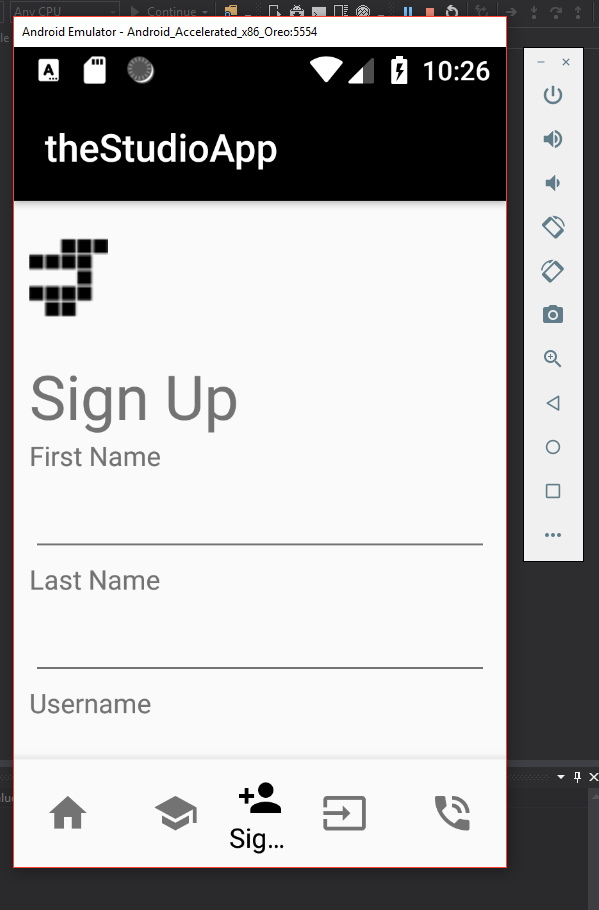
During development, virtual device debugging was not the main method I employed, this was mostly for the reasons of hardware constraints and emulation issues, the emulator would flicker intermittently with seemingly no possible fix beyond a major hardware change, in addition a large amount of memory was required to run the emulator even when debugging was not taking place, for this reason it was extremely difficult to fix issues in code due to the fact that my computer regularly became unresponsive for minutes at a time, and if I were to close it may take upwards of ten minutes to start up and begin debugging again. Regardless, here are screenshots I have taken of the application being run on a virtual device



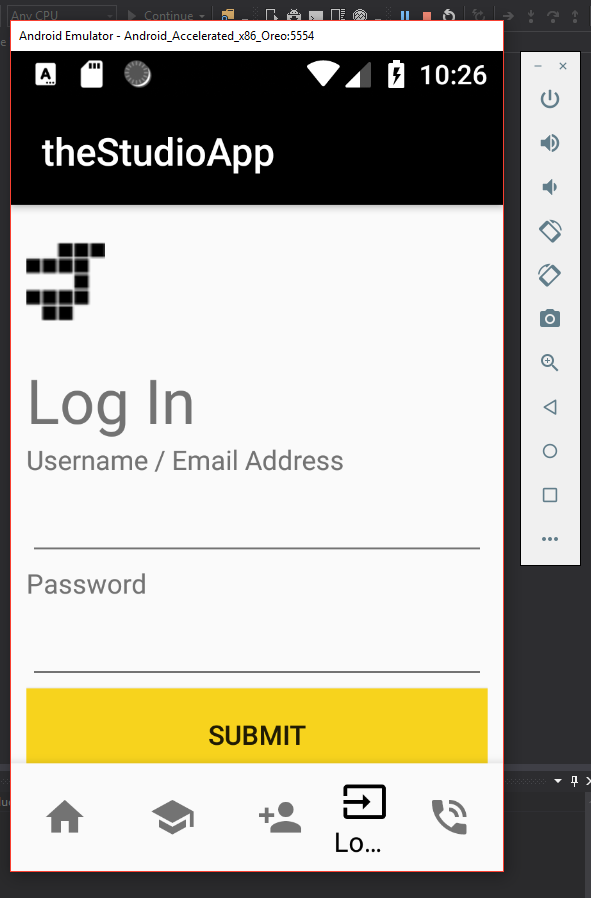
Screenshot 41, screenshot of the emulator running on the Home screen



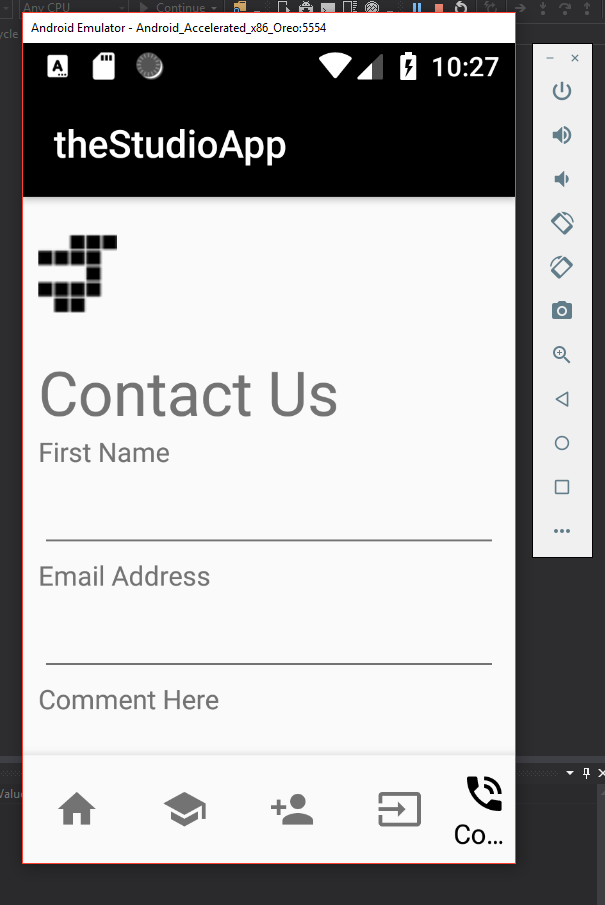
Screenshot 42, screenshot of the emulator running on the About Us screen



Screenshot 43, screenshot of the emulator running on the Sign Up screen



Screenshot 44, screenshot of the emulator running on the Log In screen



Screenshot 45, screenshot of the emulator running on the Contact Us screen

### Hardware Device

The deployment of my application to a hardware device was my main form of debugging method due to the fact that I have an android device and development was, in fact, simpler on this device than a virtual one due to the fact I am more intimately aware of the workings of my own personal device, below are photographs of my phone running my app:

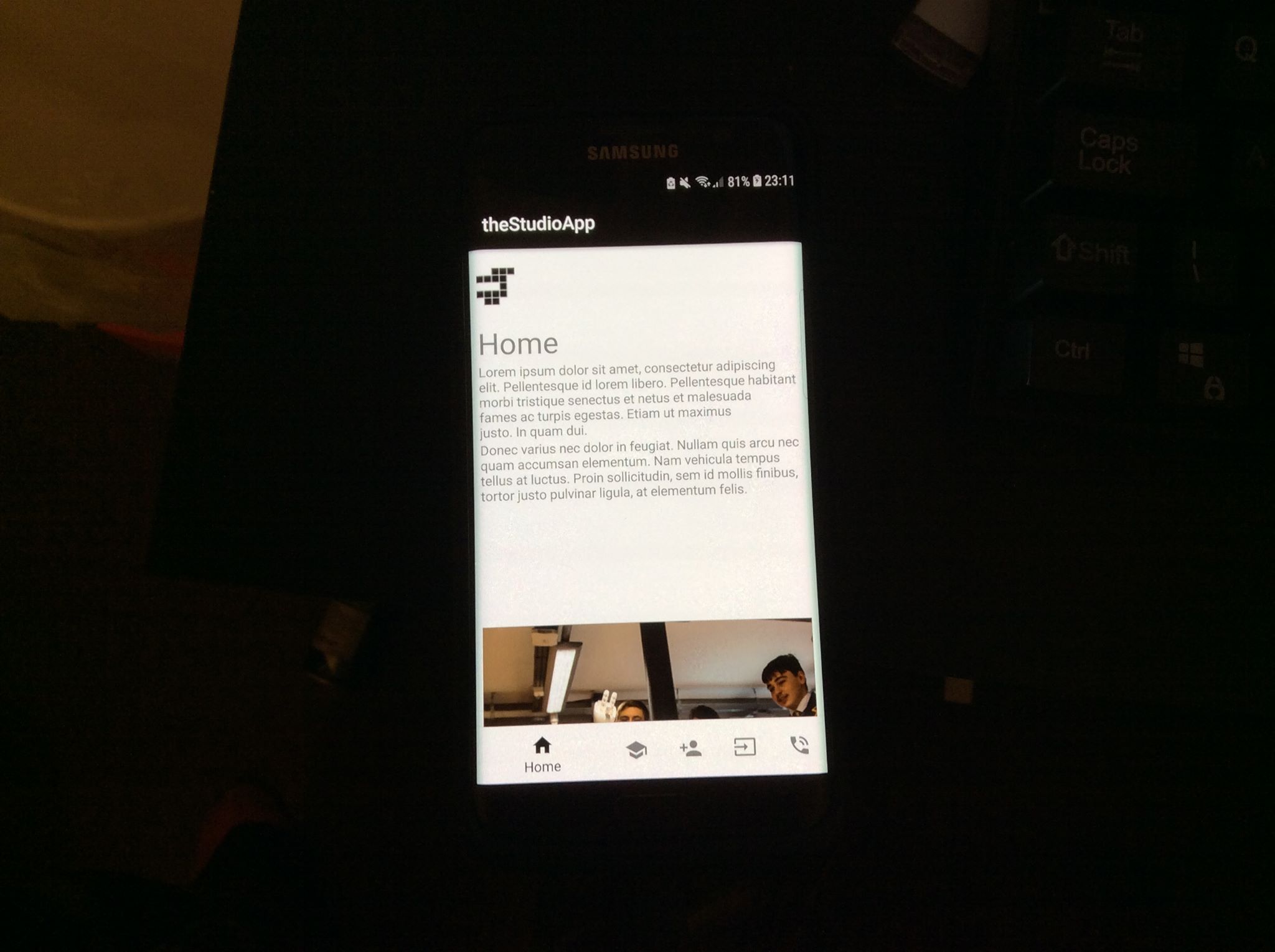


Photo 2, the Home screen running on my personal device, a Samsung Galaxy S7 Edge

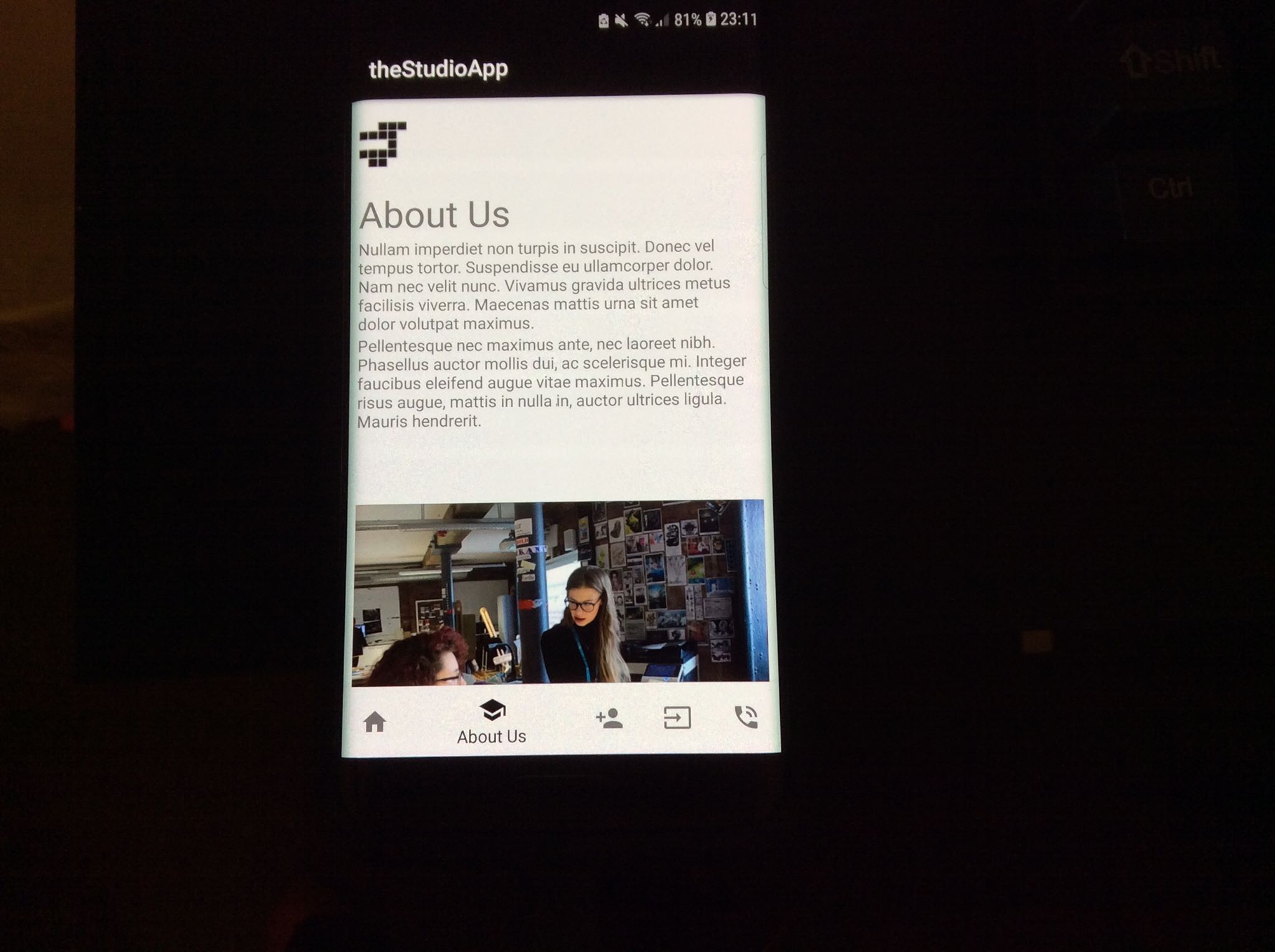


Photo 3, the About Us screen running on my personal device, a Samsung Galaxy S7 Edge

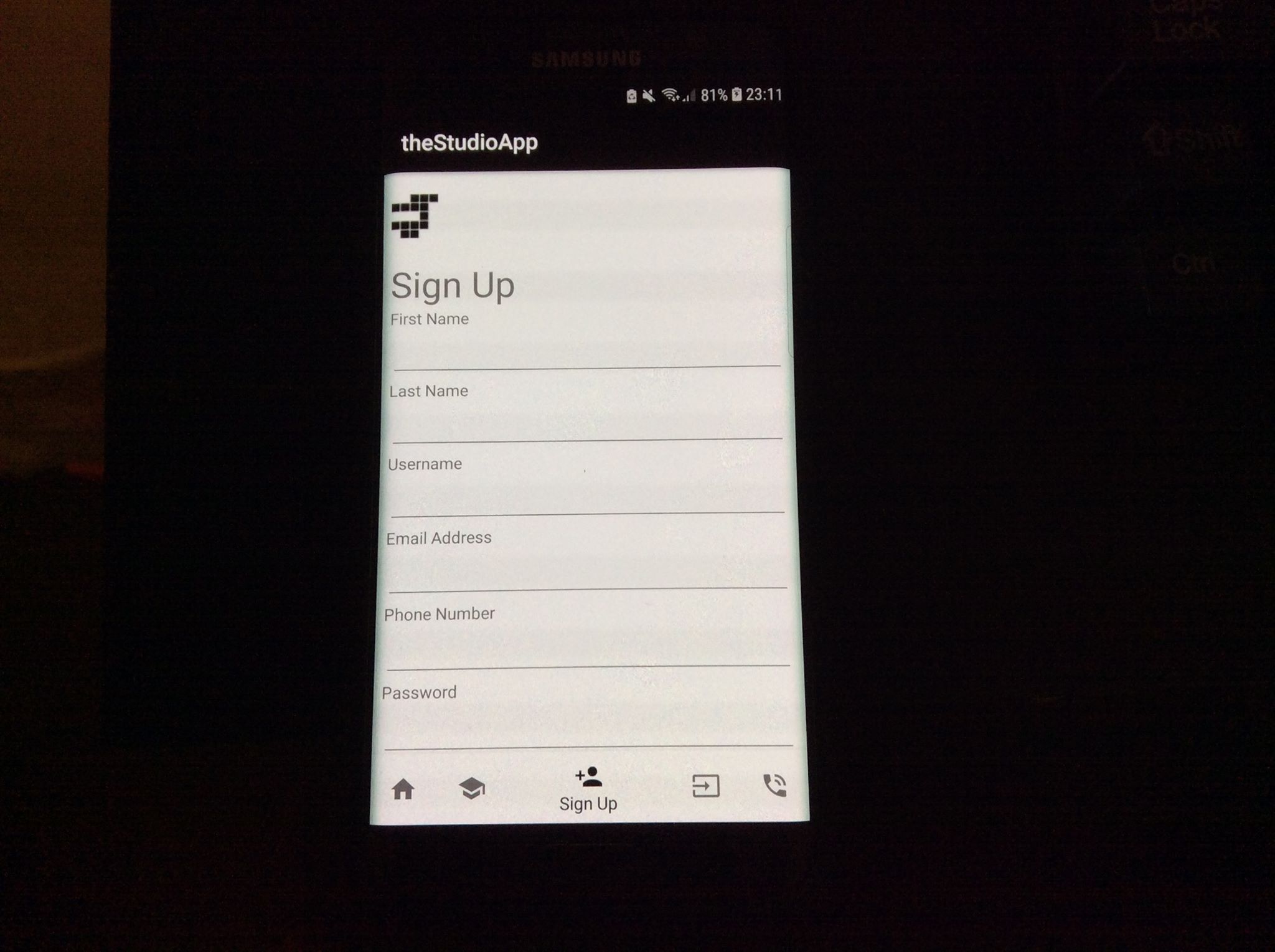


Photo 4, the Sign Up screen running on my personal device, a Samsung Galaxy S7 Edge

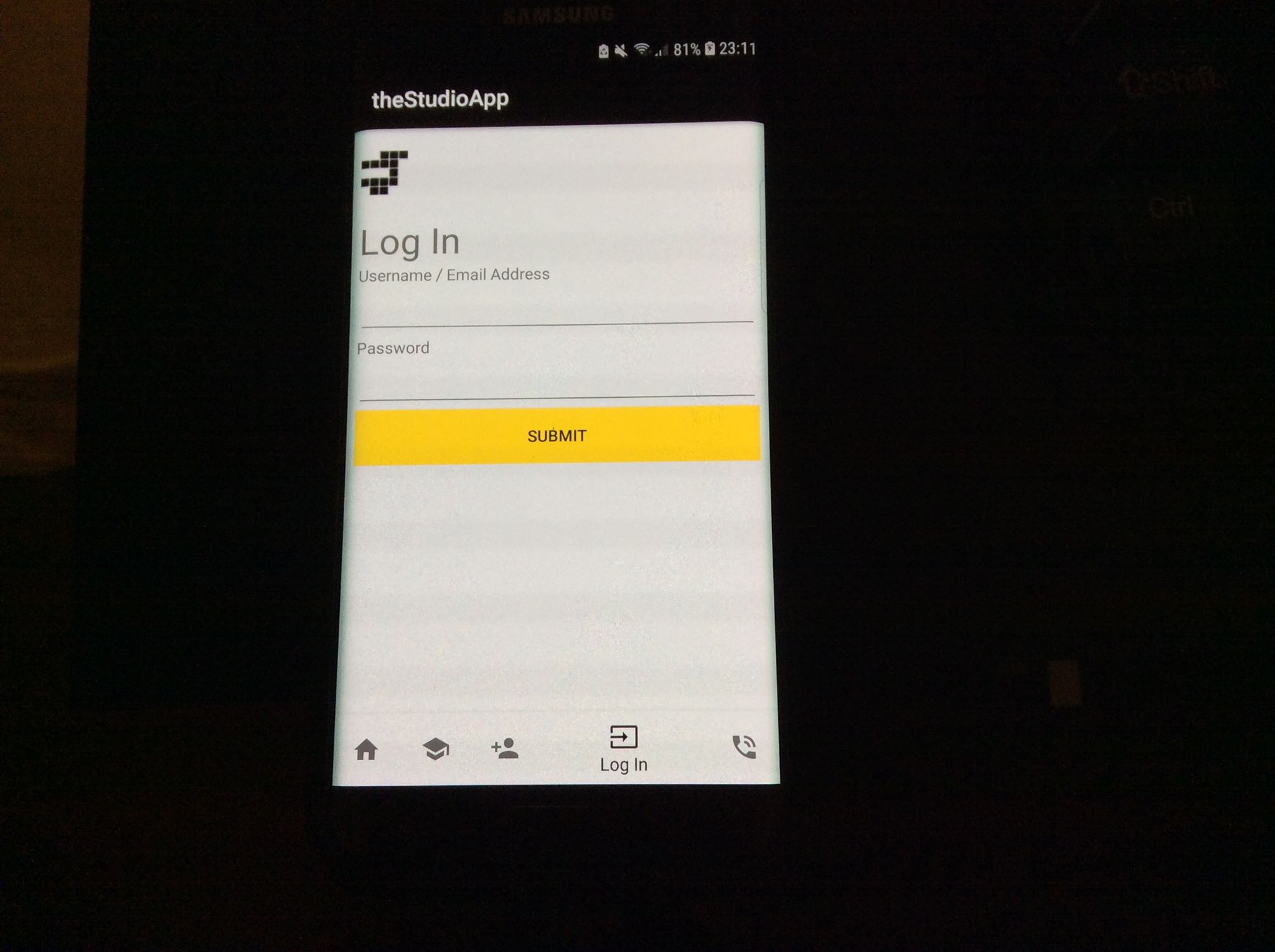


Photo 5, the Log In screen running on my personal device, a Samsung Galaxy S7 Edge

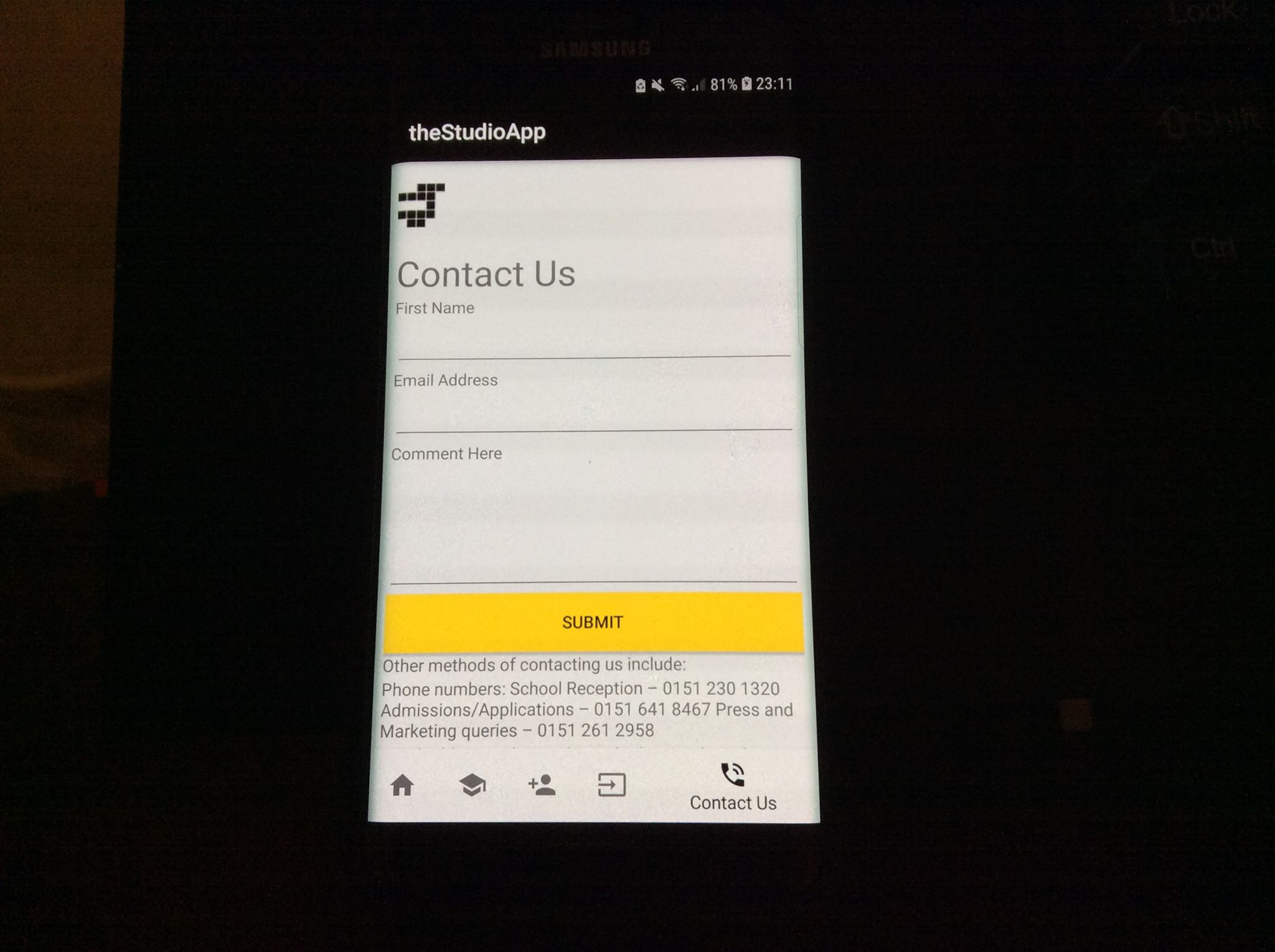


Photo 6, the Contact Us screen running on my personal device, a Samsung Galaxy S7 Edge

## Review of mobile app against client spec and customer feedback

This section of the document is dedicated to reviewing the application that has been built against the specification given at the start of the development life cycle. The specification for this app states the following: “The application should allow for both parent and student/teacher registration with account management and access to a range of services.” At this point in time this part of the specification has not been entirely met, I have allowed for simple account management through my implementation and usage of SQLite, in that a user is able to input their persona; information and, as such, use that to log in to the application.

At this point however, “A range of services” (what could be reasonably understood as dynamic content) is not available in the application. Services that are technically available are limited to client contact methods and the aforementioned account management system, a feasible update possibility is the addition of a news or information tab, wherein school news, term dates, staff lists, and other important information could be displayed in addition to dynamic content, such as ‘Hello [firstName]’ elements.

Finally, the creation of distinct parent/teacher/student registration is possible in that those who register will almost always be either parents, students, or teachers, however distinct user types are not implemented as yet, meaning there is no option for a user to sign up as either a parent, student, or teacher specifically, this is a possible feature that may be implemented in future updates.

## Evaluation of app from Client feedback and update and maintenance strategy

At this point due to the previously mentioned unfinished nature of the application it has been decided that said app will be initially released in private beta form to a select group of white and black box users for early testing and bug reporting (White Box users are familiar with the code and the inner working of an app whereas black box testers are not). It has also been decided through previously established development practices that in this early beta stage of development that updates will occur on every Tuesday of each week, with major content-adding updates occurring once every two or three weeks and the intermittent weeks being dedicated to bug fixing, a table of what is meant by this is seen below

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1st Tuesday | 2nd Tuesday | 3rd Tuesday | 4th Tuesday |
| Update | Major | Minor | Major/Minor | Minor/Major |

As one can see, the first update is major, so for example this would be the addition of the information tab, then the minor update that follows the next week may be dedicated to bugfixes and minor tweaks to content, then the next Tuesday will be either another minor update or a major one depending if there are bugs in the previous release or not and so on and so forth.

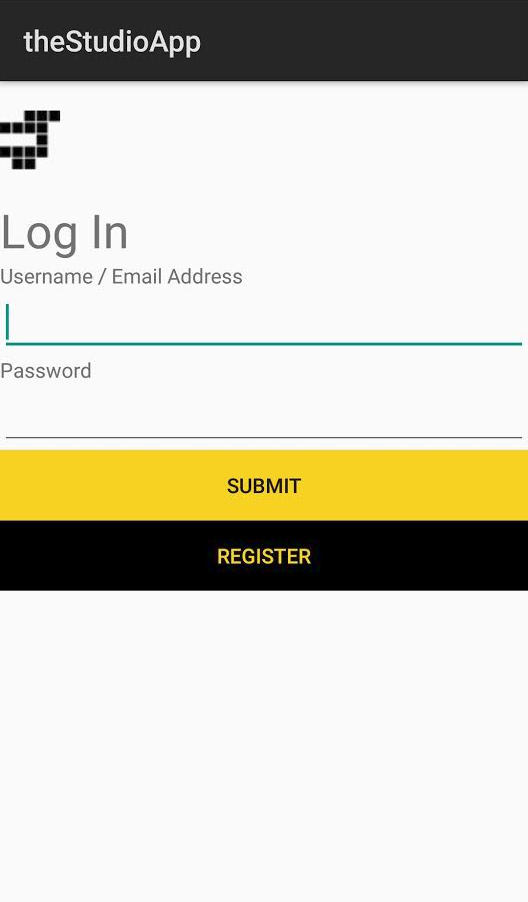
Tuesday has been picked as the weekly release day due to the fact that, firstly, preparation for the update can occur on a Monday instead of developers having to sacrifice their weekend breaks, as well as the fact that the following few days (Wednesday-Friday) can be used for debugging of the previous release.

# Documentation

## User Documentation

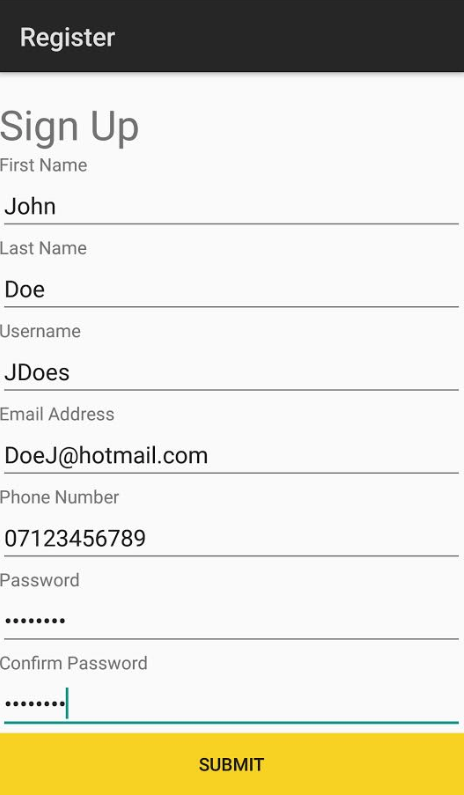
This section of the document serves as a rough help guide for an end user attempting to use the app I have developed. Luckily enough, this app is fairly simple, and as a result a large chunk of said app may just be intuitively easy to use. This is no small mistake, however, the choice to stick to a pre-cut-out path of development, that is the use of the material design philosophy, means that the user will be placed in a minor variation of what should be an extremely familiar environment. However, it must be acknowledged that this is not the case for all users, and as such this documentation must be created in order to help those individuals use the app.

Upon opening the app, the user will be greeted with the following login screen, this screen allows for the ability to use your username OR email address to log in, as well as the necessary password. This app assumes that the user is a returning one as opposed to a new one as this will be the case the majority of the time. If the user is new, in the case of a first time set up, they should press the black ‘Register’ button.



Screenshot 46, the log in screen

Upon pressing the button, the below screen will appear (without any of the example information filled in), the user is prompted at this point to put their own information in. A note: none of this information is stored anywhere other than on the device itself, this is all purely for authentication purposes only.

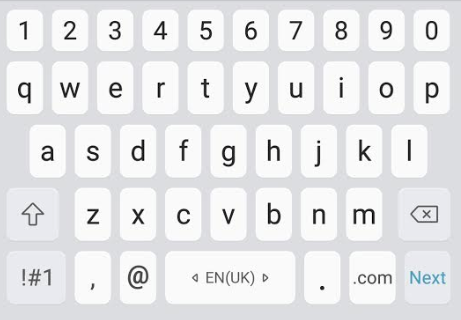


Screenshot 47, the filled-out registry form for the app

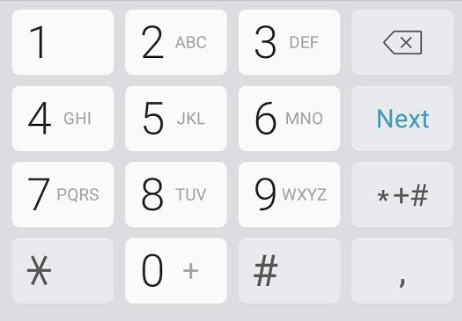
What follows are examples of variations of keyboards that are used in the app to prompt a user to put in correct information and/or make the input of specific information easier for said user.



Screenshot 48, the standard keyboard



Screenshot 49, the standard keyboard with the email variation, note the inclusion of the @ symbol in the bottom left



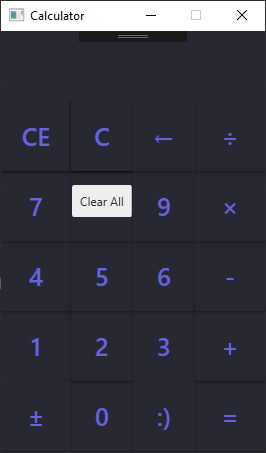
Screenshot 50, the numpad variant, for the input of telephone number

Upon the user logging in, the user experience is beyond simple, the UI at the bottom directs you to three separate pages, home, about us, and contact us, and the user can be directed to different pages by pressing the corresponding button at the bottom with their finger.

As developers we were previously planning on adding tooltips into this solution, however we decided against this as it would have drastically worsened the user experience for two main reasons.

The first of these reasons being that there was no way we could find to have a tooltip show up for a specific button the user may be confused about, say the user is confused about what the “clear” button does, on mobile every interaction is parsed as a click would be on a standard desktop or laptop setup, and as such it is not possible to differentiate between a press that is intended to find out what a button does and a press that wants to use that button. The only solution to this problem is to display tooltips for every single button, however this feeds into the second problem, that being clutter. The appearance of the application is paramount to its usability, and cluttering the screen with possibly unnecessary tooltips results in a significantly worse user experience all around.

In order to demonstrate the correct usage of tooltips, however, one may look no further than a different application I have developed, a calculator built in WPF for use on a desktop or laptop environment. The tooltip makes use of the operating systems’ inbuilt text “on hover” element, evidence of this can be seen below:



Screenshot , shows the "Clear All" tooltip being displayed after hovering on it for a short period of time (approx. 2 seconds)

## Technical Documentation

This section concerns a hypothetical future developer who may not have any prior knowledge of the app and is tasked with picking up development where it has been left off previously.

Below is an image of the full solution explorer window for the project, including the several nested folders therein. What follows is an explanation of each of the folders and a selection of the files within.

The app is written in Xamarin.android and makes use of the XML language for frontend and the C# programing language for logic with using SQLite; for backend functionality, a developer must rigidly stick to the material design methodology (<https://material.io/design/>) when building this app in order to integrate it into the android OS seamlessly.

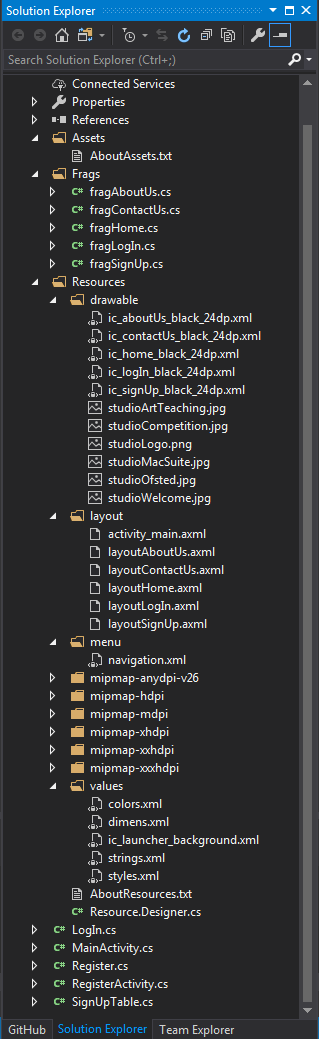


Figure 1, the solution explorer for the app

### Assets

At present this folder is not in use, this is primarily due to the presence of the “Drawable” folder’s existence. Within the AboutAssets.txt file, the only file in that folder, is this message:

Any raw assets you want to be deployed with your application can be placed in

this directory (and child directories) and given a Build Action of "AndroidAsset".

These files will be deployed with you package and will be accessible using Android's

AssetManager, like this:

public class ReadAsset : Activity

{

protected override void OnCreate (Bundle savedinstanceState)

{

base.OnCreate (savedinstanceState);

InputStream input = Assets.Open ("my\_asset.txt");

}

}

Additionally, some Android functions will automatically load asset files:

Typeface tf = Typeface.CreateFromAsset (Context.Assets, "fonts/samplefont.ttf");

It is not beyond the realm of feasibility that a future developer may decide to migrate any assets from their presently existing folders into the Assets folder however this may not achieve anything other than being aesthetically pleasing and present a further ease of understanding to any future developer.

### Frags

The Frags, or Fragments folder, houses the code that is required to deploy the fragments for each page of the app, the following code example is from the fragAboutUs.cs file and is essentially identical from the rest apart from the highlighted sections, which change depending on the page each frag is calling:

namespace theStudioApp

{

public class fragAboutUs : Fragment

{

public override void OnCreate(Bundle savedinstanceState)

{

//notifies when fragments are run

base.OnCreate(savedinstanceState);

Toast.MakeText(Context.ApplicationContext, "fragAboutUs oncreate has been run", ToastLength.Short).Show();

}

public override View OnCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedinstanceState)

{

//gets the layout file and makes it a view, inflates it to the size of the screen

return inflater.Inflate(Resource.Layout.layoutAboutUs, container, false);

//TODO: remove unreachable code below

//return base.OnCreateView(inflater, container, savedinstanceState);

}

}

}

### Resources

This folder has the majority of content within the app nested within folders herein and as such is arguably the most important folder, making changes (specifically removing items) here is not advised whatsoever.

#### Drawable

This folder contains the icons and images used for display within the app. For the most part the images herein are .jpg files, however where necessary (for the proprietary Studio logo) .png files are used.

However for icons, specifically the material icons used for navigation in this instance, must be encoded as an .xml file in the svg, scalable vector graphic format. This is to ensure graphics do not lose quality if or when they need to be scaled. What follows is the full svg formatted .xml file describing the home icon, with comments

<vector xmlns:android="http://schemas.android.com/apk/res/android"

android:width="24dp"

android:height="24dp"

android:viewportHeight="24.0"

android:viewportWidth="24.0">

<!--

in pathData

M/m moves cursor to position, always followed by x,y coords

Z/z draws a line to current position of cursor to start position

L/l draws line from current position to position specified by x,y

H/h draws horizontal line from current to position specified by x

V/v draws vertical line from current to position specified by y

Uppercase = absolute, lowercase = relative

-->

<path

android:fillColor="#FF000000"

android:pathData="M10,20v-6h4v6h5v-8h3L12,3 2,12h3v8z" />

<TextView

/>

</vector>

#### Layout

Written in XML to serve as the front end of the app, contains code for buttons, text boxes, images, and text. What follows is the code for the smallest page, the login page

<?xml version="1.0" encoding="utf-8"?>

<!-- the whole page is nested in a scrollview so the page can be scrolled if the

amount of content on the page doesn't fit the screen its displayed on -->

<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<!-- linear layout describes a visual layout that is linear, i.e. one

after another -->

<LinearLayout

android:orientation="vertical"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<!-- the studio logo as called from the drawable file -->

<ImageView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:src="@drawable/studioLogo" />

<!-- text saying 'log in' as called from the strings file -->

<TextView

android:id="@+id/title\_logIn"

android:layout\_height="wrap\_content"

android:layout\_width="wrap\_content"

android:text="@string/title\_logIn"

android:textSize="32sp" />

<TextView

android:id="@+id/userName"

android:layout\_height="wrap\_content"

android:layout\_width="wrap\_content"

android:text="Username / Email Address" />

<!-- an input box for the username and/or email address -->

<EditText

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:id="@+id/userNameEmailAddress" />

<TextView

android:id="@+id/password"

android:layout\_height="wrap\_content"

android:layout\_width="wrap\_content"

android:text="@string/password" />

<EditText

android:inputType="textPassword"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:id="@+id/txtPassword" />

<Button

android:text="Submit"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:id="@+id/btnSubmit"

android:background="@color/colorAccent" />

<Button

android:text="Register"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:id="@+id/btnRegister"

android:background="@color/colorPrimary"

android:textColor="@color/colorAccent" />

</LinearLayout>

</ScrollView>

#### Menu

This folder houses the navigation functionality, i.e. the menu, in the application, all of the content pages pull from this menu and frags are run based off of what are essentially onclick event handlers from this menu system.

What follows is the code for the menu, it contains three items, home, about us, and contact us, and are labelled as such

<?xml version="1.0" encoding="utf-8"?>

<menu xmlns:android="http://schemas.android.com/apk/res/android">

<item

android:id="@+id/navigation\_home"

android:icon="@drawable/ic\_home\_black\_24dp"

android:title="@string/title\_home" />

<item

android:id="@+id/navigation\_aboutUs"

android:icon="@drawable/ic\_aboutUs\_black\_24dp"

android:title="@string/title\_aboutUs" />

<item

android:id="@+id/navigation\_contactUs"

android:icon="@drawable/ic\_contactUs\_black\_24dp"

android:title="@string/title\_contactUs" />

</menu>

#### Mipmap

The Mipmap series of six folders serve to house increasingly sized versions of the app’s icon, as seen below:



Figure 2, theStudioApp's icon as will be seen on the user's phone, sourced from the folder 'mipmap-xxxhdpi'

Five of the six folders (mipmap-mdpi – xxxhdpi) house .png files of the icon itself (as well as the rounded variant), with the anydpi folder serving to house references to colour values the icon holds, this is as seen below:

<?xml version="1.0" encoding="utf-8"?>

<adaptive-icon xmlns:android="http://schemas.android.com/apk/res/android">

<background android:drawable="@color/ic\_launcher\_background"/>

<foreground android:drawable="@mipmap/ic\_launcher\_foreground"/>

</adaptive-icon>

#### Values

The values folder contains, as the name suggests, all of the values required for use in the app. As previously explained, the way this implementation of Xamarin works is instead of having content such as colour values, text, and other stylings written inline or in a single, separate file that explicitly corresponds to a specific section of the app (Like with HTML/CSS), all values are declared in a separate folder with different files representing different uses, and are then referenced by a declared name elsewhere in the program.

As a simple example, say a developer wants to display the word ‘Home’ as a title in one of the screens in their app, instead of:

<TextView

android:id="@+id/title\_home"

android:layout\_height="wrap\_content"

android:layout\_width="fill\_parent"

android:text="Home"

android:textSize="32sp" />

They would instead write:

<TextView

android:id="@+id/title\_home"

android:layout\_height="wrap\_content"

android:layout\_width="fill\_parent"

android:text="@string/title\_home"

android:textSize="32sp" />

And in …\values\strings.xml we would see:

<string name="title\_home">Home</string>

### Miscellaneous and Other

The final set of files occupy space outside of folders within the root of the solution, these files, from a development standpoint, are ideally not to be altered as they form the basis for several crucial basic tasks of functionality, such as frag switching and logging in, that should be accessed as quickly and as easily as possible by any other section of the application.

# References

Android Developers, 2018. *Sign your app.* [Online]   
Available at: https://developer.android.com/studio/publish/app-signing  
[Accessed 9 September 2018].

Android Source, 2018. *Application Signing.* [Online]   
Available at: https://source.android.com/security/apksigning/  
[Accessed 9 September 2018].

McLemore, M., Opgenorth, T., Dunn, C. & Umbaugh, B., 2018. *Signing the Android Application Package.* [Online]   
Available at: https://docs.microsoft.com/en-us/xamarin/android/deploy-test/signing/?tabs=vswin  
[Accessed 9 September 2018].

Wagner, B. et al., 2015. *C# Coding Conventions (C# Programming Guide).* [Online]   
Available at: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-convention

1. Note, a mistake was made with the bottom text box, it has been rectified to resemble Screenshot 2 exactly. This is reflected in the UI element grid [↑](#footnote-ref-1)
2. This page and those below do not include a submit button, this has been rectified in more recent screenshots and reflected in the grids below [↑](#footnote-ref-2)