Table of Contents

Analysis

The Problem

The problem I am trying to tackle, in a nutshell, is this: the current widespread methods of tracking personal goals, habits and projects do not have reliable methods for keeping their users accountable for their progress.

The two most popular methods of tracking personal goals in widespread use are:

1. To use a physical journal
2. To use a piece of software

The journal has the least input when it comes to getting the user to stick to their goals. It is an inanimate object with no automated functionality. It cannot tell its owner to sit down and write out their goals for the day, nor can it remind them when they need to log their progress.

Goal/habit tracking software is a bit better than the journaling method at getting their user to follow through on their targets. Through the use of push notifications and alerts, the software can remind its user of the goals they have set and alert them as to when they need to log their progress. The problem is, this method alone is not enough to make someone consistently work towards a target or build a habit. The reason why this is not enough is because the user feels no connection to the notifications or the software they are using; if they do not follow through with their pre-set goals, there are no consequences. The person using the software has no desire nor any want to please or placate a computer program and so, whilst notifications are useful reminders, the motivation required to work at a long-term project or target is entirely dependent upon the user.

The journal suffers the same drawback. The owner of said journal is not emotionally involved with the piece of paper to which their plans and hopes have been laid bare. There is no sense of urgency to complete their goals and log their progress because there are no consequences exacted by the journal if they are idle in their endeavours.

My proposed solution to this problem is to create a mobile application with a method of keeping the user accountable for their progress, or lack thereof, by sharing their endeavours with something that has an opinion that people care about: other people.

Users of this app will be able to set a short or long-term goal (which I shall call a project) for themselves which they can then link to an "accountability group". This accountability group is, in essence, an online group chat which consists of other people who are all working towards their own goals. Each member in the group shall be able to see each other's progress. Should one member fail to meet one of their targets, or at least fail to record their progress, a message shall be posted in the group chat alerting the other members.

These accountability groups provide the user with a team of people they can help reach their goals and encourage and from whom they themselves can receive encouragement. The burden of trying, day-in, day-out, to meet your targets has now been somewhat lessened as your family, friends and colleagues help you carry it and, when the inevitable stumble happens and you lose your drive and desire to climb the mountain of your project, those supporting you are reason enough to get back up and continue on.

Solving the Problem by Computational Methods

I believe the use of computational methods in the form of a mobile application is best suited to solving the issue I have described above for three reasons:

1. Mobile phones are the best platform for electronic communication
2. Software can be used to automate repetitive tasks
3. Mobile phones are the incredibly portable

My solution relies on the use of group chats to keep users accountable to other users. Mobile phones are the most used platform for quick electronic communication and so naturally they lend themselves to this solution. Through the use of a mobile application, the user will be instantly updated on the progress of the other members in their groups and the other users will instantly receive updates on the progress of the user, making sure everyone is kept accountable. These live updates would not be possible just using a standard paper journal as it does not come with a telepathic connection to other journals.

The use of a computer program over a physical method of goal tracking saves the user time as they don't have to format anything or repeatedly write anything. If you wanted to track your daily goals in a journal for instance, you would have to set out each entry by hand each day. With an app, everything is already formatted for you and, if you have a goal you want

to set yourself on a regular basis, the software can automatically set the same goal to repeat after a specified time period has passed. The automation possible with a computer cuts down on boring repetitive tasks, allowing the user to get on with the activity that really matters without distraction.

The portability a mobile phone also lends itself to the solution of this problem. If you were to solve this problem, a lack of accountability, via, shall we say, traditional means, you would have to meet up with a group of friends daily and ask how they were progressing towards their targets. Whilst in an ideal world this would be the case, in the world that we live in people are generally quite busy. There may not be enough time for you and your friends to discuss how you're doing over a cup of coffee each day. This physical version of the aforementioned accountability group also excludes those who do not live within a few miles of your chosen "hang-out". What if you have a friend who lives on the other side of the world whom you would love to have in the group?

The use of a computational method completely resolves these problems. You may not have enough time to meet for an hour with a group of people on the daily, but most people have no problem with taking 10 seconds to respond to a message. The use of an online group chat means you don't have to regularly meet up because your friends are always in your pocket!

Due to the extreme portability and convenience of being able to access the internet at all times, the majority of the population carries around their mobile phone for almost the entire day. Using a mobile app instead of a physical means to track your goals therefore eliminates the excuse of not having enough time to log your progress and/or you went and left the notebook you use to track your targets at home. You can log you goals and be reminded of your targets anytime, anyplace.

The Stakeholders and the Role They Will Play

This app is targeted at those who have reached the understanding that goal and planning is the foundation of progress, but it is not exclusively made for those above a certain age. The hope for this app is that anyone with a serious desire to achieve a goal, complete a project or form a habit will be able to quickly form an online group of people with a similar drive to maximise their chances of success. The app is not made for a specific age bracket.

The app is targeted at those who want to set personal, study or work-related goals and so I have a large array of potential stakeholders. To cover the school aspect of things, I shall consult with some of my peers at sixth form to glean any potential design features I could include to help improve the functionality of my app. My dad is the owner of several small companies and so I shall refer to him for ideas about how the app can be made applicable to those in work and also how it could potentially be used as a communication and project outline platform among employees working together on the same project. Friends of varying ages will also be able to provide feedback on how the app could be designed to maximise productivity when it comes to pursuing a personal goal or trying to form a habit.

Other Systems and Ideas

Because my project is a combination between both a goal tracker and a personal messaging app, I've looked at and tried a few of each. Below are some screen shots and a summary of my thoughts on the apps I have tried.

Strides

Figure 1.

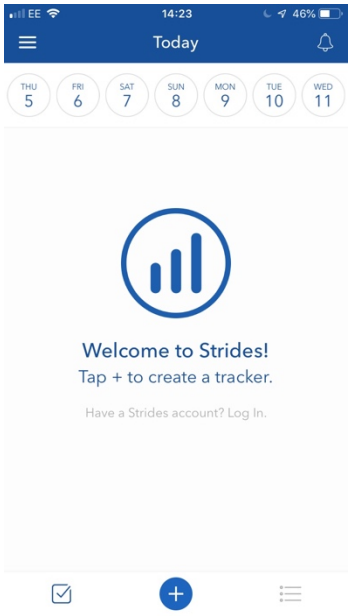


Figure 2.

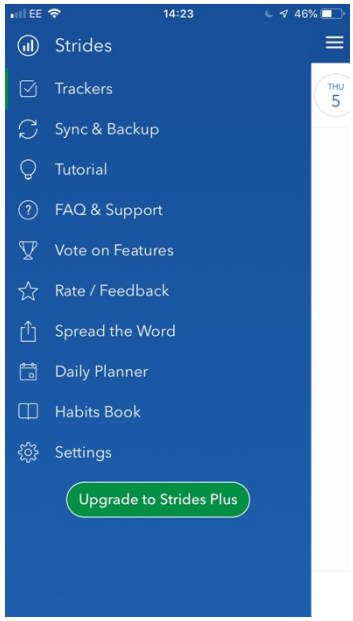
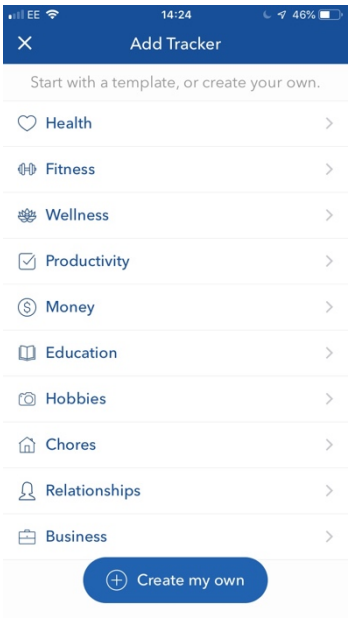


Figure 3.



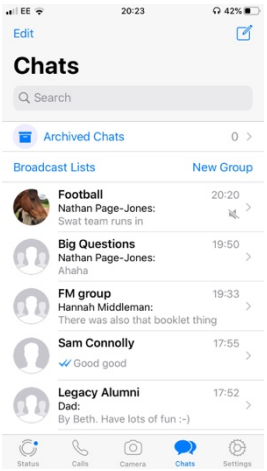
This app is about as good a goal tracker you can get. It's simple to use, and incredibly flexible. When you open the app you are greeted with the screen shown in Figure 1. The layout is fairly intuitive and you immediately know how to add goal. Having said that, I wasn't entirely sure what the two icons flanking the "+" did until I used them so that is something I can improve on.

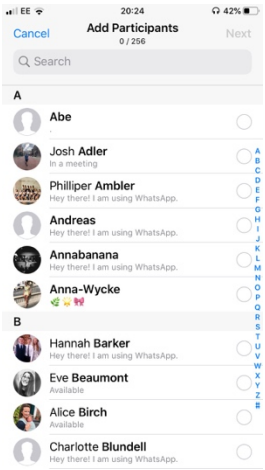
Figure 3. shows the screen that pops up when you click the "+" to add a goal. It comes with an extensive pre-set list of "Trackers", which, in my opinion, was unnecessary as the time required to setup your own custom "Tracker" was not far removed the time it took to find and modify the pre-set "Tracker" that matched the habit/goal you wanted to track.

When making your own custom goal you could pick one of three tracking methods. You could either track it as a habit, where you either succeed or you didn't; as a number, where you could either try and reach a target total by a certain date or track your average number over an amount of time selected by you; or as a project complete with milestones. I loved the ability to choose out of several tracking methods as it gave me an incredible amount of flexibility when it came to the habits or goals I was trying to track. You could log your progress in anything.
My favourite method for goal tracking from the app however was the ability to set long term projects that you split into smaller milestones. It is the best way that I have come across for tracking long term goals.

The one thing I thought this app was lacking was a sure-fire way to keep their users accountable. I myself, though I thought it a brilliant application, felt no compulsion to actually try and persist in my daily habits, particularly when I became busy with school work, because the app gave me no reason to.

WhatsApp





What better messaging application to draw inspiration from than WhatsApp? As of July 2019, it is the most popular mobile message app in the world with an average of 1.6 billion users accessing the application on a monthly basis.

WhatsApp is an incredibly simple app to use and communicate. One of the biggest selling points of the app is that you don't have to set up an account with a separate WhatsApp username and ID to message people and you don't have to create a new contacts list on the app. Straight from opening the app, you can message anyone using WhatsApp so long as you have their phone number within your OS's default contacts list. I think this is one of WhatsApp's best features as it means you don't have to waste time go collecting user IDs from all you existing phone contacts to create a new app-based contact list.

The group chat system in WhatsApp is also incredibly simple to use. You tap "New Group" on the "Chats" screen, select people from your contacts whom you wish to include in the group, give the group a name and voila, you have your group chat.

One of the few criticisms I have about the app is its lack of an interesting colour scheme. For the most part, the app consists of white and varying shades of grey with the odd light green highlight thrown into the mix. It's rather uninspiring.

How My Research Will Affect My Design

The "Strides" app has shown me just how important flexibility is to the user when it comes to tracking goals. Giving the person using your app the ability to choose between several different methods when it comes to tracking their own personal goals gives them much greater freedom when it comes to deciding how and what to track, increasing the odds of their tracking methods and goals being much more suited to the user's needs. As a result, I will incorporate the three main methods of goal tracking available in Strides within my own app, allowing the user to measure their progress with the traditional success/failure approach along with both a numerical and project-milestone alternative.

Based off of my research into WhatsApp, I will incorporate a similarly painless contact experience in my own app by allowing the user to set up accountability groups with people in their own contacts list. By removing the pain of creating a contacts list from my app experience, I hope to encourage people to fully utilise the app's accountability groups. Having said that, it's worth bearing in mind I don't know, as of yet, whether it would be easier and more efficient from a development perspective to simply get the user to create an account and use a computer-generated ID instead of pre-existing phone numbers. It is something I shall review during development.

The Limits of the Project

One of the limits of the project is that success, at the end of the day, is ultimately up to the user. Whilst the solution I have proposed helps alleviate some of the burden placed on the shoulders of the user by sharing it round with a group of people it is still up to them whether they make the effort to peruse their goals.

Another limitation of the project is the amount of time I have for development. I have the space of a few months to plan, design and build a fully-fledged group goal tracking app with messaging capabilities. My limited experience with app development, with this being my fist major project, buts a box around the functionality of the app that I won't be able to get out of within the space of a few months.

The form my proposed solution shall take is that of a mobile app. However, there are multiple operating systems out there for smart phones, each with their own development languages and techniques. Because I am a one man team, I chose to only focus on developing my app for Apple's iOS as it is the ecosystem I am best equipped to work with. As a result, my solution will not be available for those who do not own an iPhone.

I shall also only be developing this application for mobile phones. Tablets and desktops will not be able to run this application. Whilst I may only have a little experience with developing iOS apps for the iPhone, I have no experience developing applications in the Swift language for tablets or desktop computers. To save time trying to learn how to write applications that will work across multiple devices I have decided to stick with making an app that exclusively operates on an iPhone.

Project Requirements and Success Criteria

Hardware requirements:

- An iPhone that can run iOS 13
- X amount of memory

If the above hardware requirements are met, the user should be able to run my app.

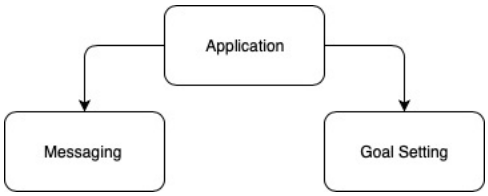
Success Criteria:

- The user can create and track:
    - A goal
    - A habit (a regular short-term goal)
    - A project
- The user can successful send and receive messages to and from a group chat
- The user can link a goal to an accountability group

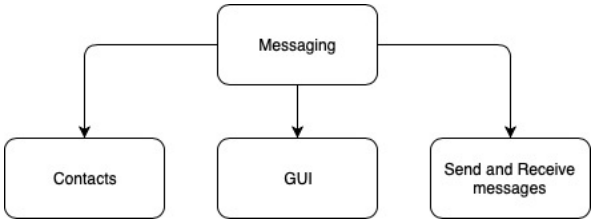
Design

Breaking Down the Problem

Straight off the bat, this problem can be broken into two components: the goal setting and the messaging. The user needs to be able to set goals and then link them to a group chat. For this to work, I need a fully functioning messaging system that draws on some form of contact list to display, store, send and receive messages along with a section of the app that allows you to actually create and track your goals. For all intents and purposes, these two components can be developed individually with only a little bit of code being necessary to tie them together.

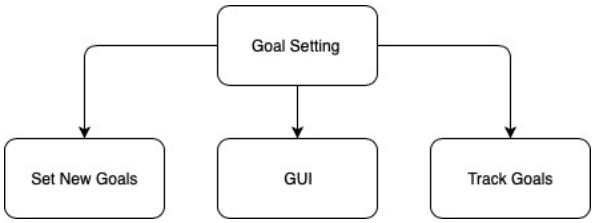


Each of these components can also be broken down similarly into several distinct sections. For the messaging component, there are three distinct areas:



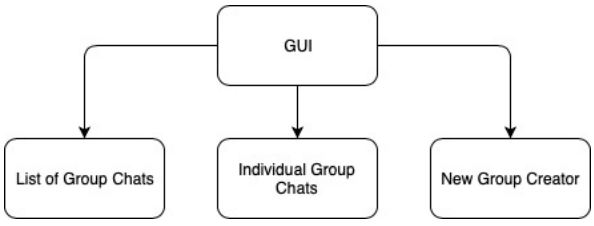
I will need to have the app access some form of contacts list, whether it is the user's pre-existing set of contacts stored on the phone or a list created especially for use within the app. Along with a set of contacts, I'm going to need a whole GUI that allows the user to view received messages as well as group chats. Perhaps the most important component in the Messaging section is the ability for the user to both send and receive messages. To do this, I'll use Apple's Messages and MessagesUI libraries.

For the goal setting part of the app, there are also three distinct components:



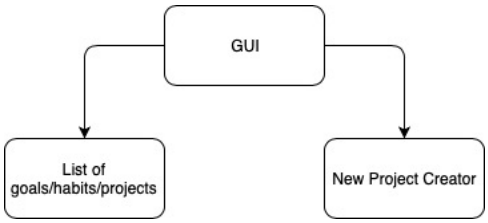
The user needs to be able to set goals/targets/habits. The solution will have to include a method for setting a new goal as well as a way to store and retrieve goals once they have been set. All this information will be shown to the user via a GUI which will have to incorporate both these components.

When making the GUI for the messaging part of the application, there are three screens minimum that I'm going to have to create:



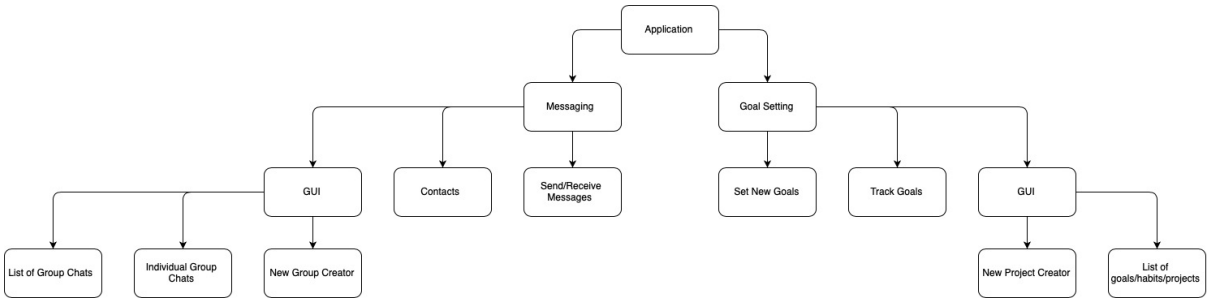
The user will need to be able to view all the groups that they are part of in some sort of list. When they click on a particular group, they will want to see the messages they and others have sent to that group. If a user wants to create a new group, there will also have to be a screen that allows them to add name and members.

The goal setting part of the application will also have its own set of screens:



There will need to be a display showing the goals set by the user that they wish to track along with some screen that allows the user to create new goals.

So far, a basic breakdown of the program looks like this:



Let's dive in a little deeper.

The Structure in Detail

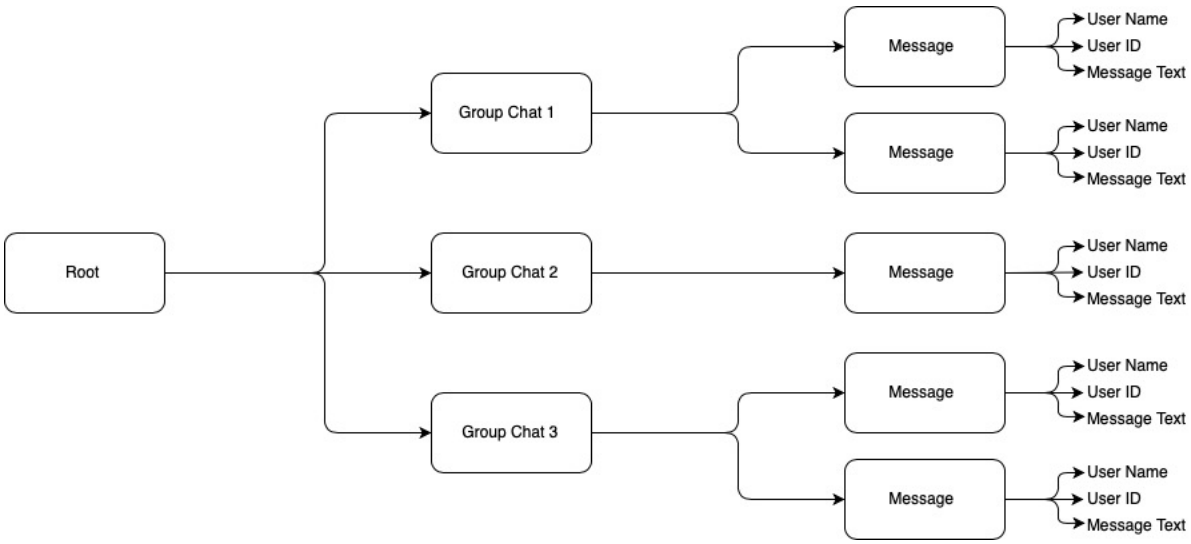
The Messaging Component

Overview

The first problem we are going to need to solve is how to send and receive messages between different users. One solution to this conundrum, and the solution I intend implement, is to set up an online database.

When a user sends a message, it will be uploaded to a branch on the database which represents the selected group chat. The contents of the message, as well as the sender's name and unique ID will be pushed to the group chat branch of the database. The recipient will automatically pull down any new messages from the online database when they click on the group chat. For my online database, I will be using an online Google service called Firebase which will allow me to push and pull new messages in real time.

The database will be structured very much like a tree:

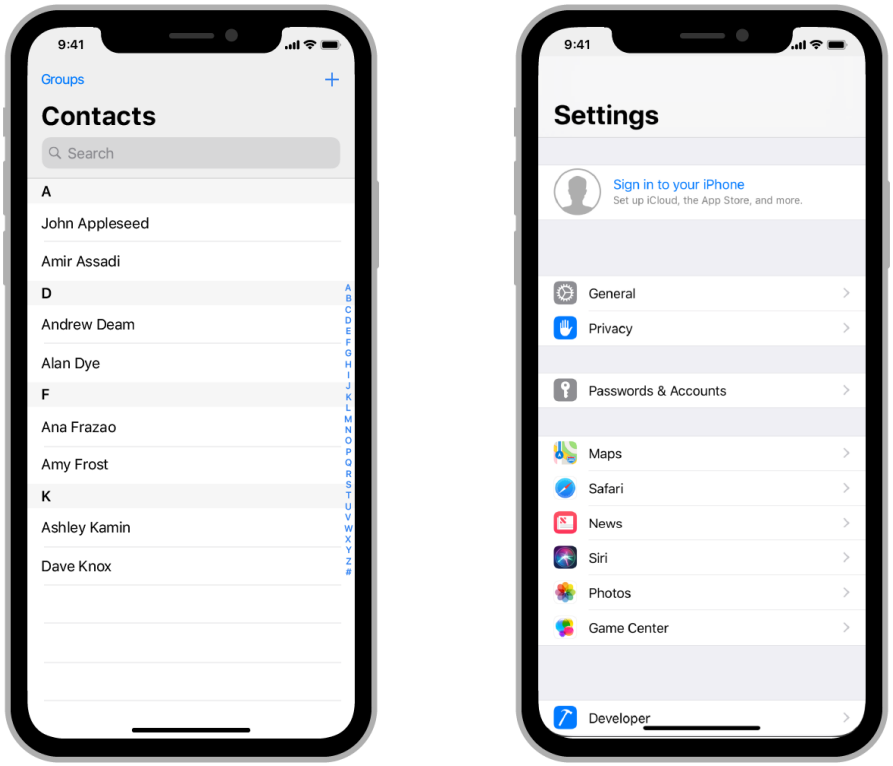


Group chats will simply be nodes on the tree and the user can access different group chats by changing the branch of the database they are accessing based on which group chat icon is selected. Each message will store the name of the user who sent it, that user's unique ID (whether it be auto generated, an email address or a phone number) and the message content.

UI

For the UI, I will use a class from a third-party library called JSQMessagesViewController that comes with all the necessary methods to quickly create a chat UI. All I will have to do is create a subclass that overrides the parent methods so I can tailor the chat for my own purposes.

The group chats will be arranged in a "Table View", something I can implement using Swift's UITableView class.



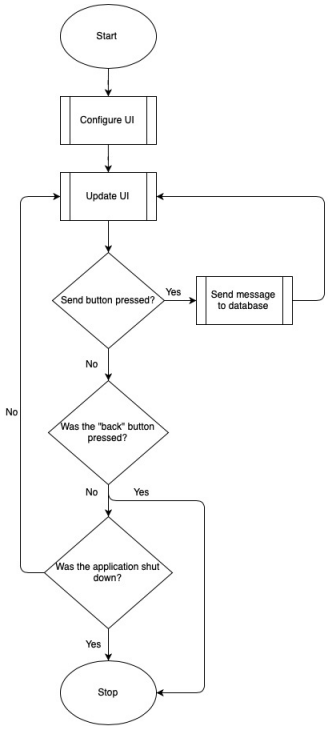
Above is an example of a table view.

In the example of the Contacts app in the above image, each cell within the table, such as the one containing the name “Andrew Deam”, can be clicked on. When the user clicks on a contact, they are taken to a new screen that contains that contacts information, such as phone number, address etc. This is how the group chats shall be displayed in my own application. When someone clicks on a cell in the table of group chats, a function will update the current branch

When you move to the messaging component of the app, you will be greeted by a list of the groups you are a part of. When a particular group is selected, the database path will be changed accordingly to make sure the messages from the correct group will be displayed.

Sending and receiving messages

When the user presses the “send” button in the UI, the software will execute a function that takes the message data, username and user ID of the sender and upload that information to the correct branch of the database. As soon as those changes are made, the applications on the mobile devices of other users will pull down those changes. Below is an example of how the group chats will work:



Let's take a look at what's happening here:

1. Configure the UI settings so that the correct group chat is displayed and the message bubbles and text labels are set to the right size, shape and colour.
2. Update the UI to display all recent messages.
3. Check if the "send" button has been pressed.
    - If it has, run the "Send message to database" sub-routine and return to step one.
4. Was the "back" button pressed?
    - If it has been pressed, return to the previous screen and end the messaging algorithm.
5. Was the application shut down?
    - If the answer is yes, end the messaging algorithm.

This is an abstracted version of what I will actually have to write but the flow chart represents the overall flow of a chat component.

The "Update UI" sub-routine will contain code that checks if any changes have been made to the database and, if so, update the UI to display those messages.

The Goal Tracking Component

Overview

There are three major things the goal tracking component is going to need to do:

1. Display the user's current goals
2. Update the goals on display when a user completes a goal
3. Update the goals on display and send both the user and accountability group an alert if the user fails to achieve the goal.

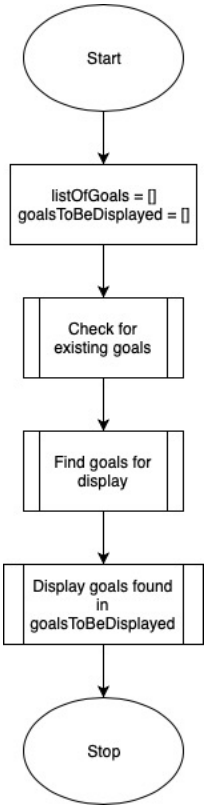
I will keep track of both the user's current goals and the goals to be displayed with the help of arrays. Each goal will be an instance of a structure I will create called "goal" (a structure, in Swift, is very similar to a class but it doesn't utilise inheritance). The goal structure will contain information about the goal such as its name; whether it's recurring; whether it has any milestones and, if so, what they are; when the goal is due to be completed; when the goal is next due to be displayed and the name of the group chat the goal is linked to.

```
structure Goal:
      var name: String
      var recurringInterval: Int
      var mileStones: Goal = []
      var dateDue: Date
      var nextDateToBeDisplayed: Date
      var linkedGroupChat: String
```

I'm not entirely sure if I will actually need to have a nextDateToBeDisplayed variable but I figured I would include it just in case it was useful.

On Start Up

When the user opens the app, there are a few things that have to happen. First, the software needs some way of tracking the user's goals and which of those goals needs to be on display. The software, when it boots, will need to check if the user has a previously existing goals and, if so, retrieve them from storage. This information could be stored on the user's iPhone or on the online database under a separate node called "users". Currently I am leaning more towards using an online database to store a list of users as this will allow people to store their account information online. Below is a flowchart displaying the start-up process.

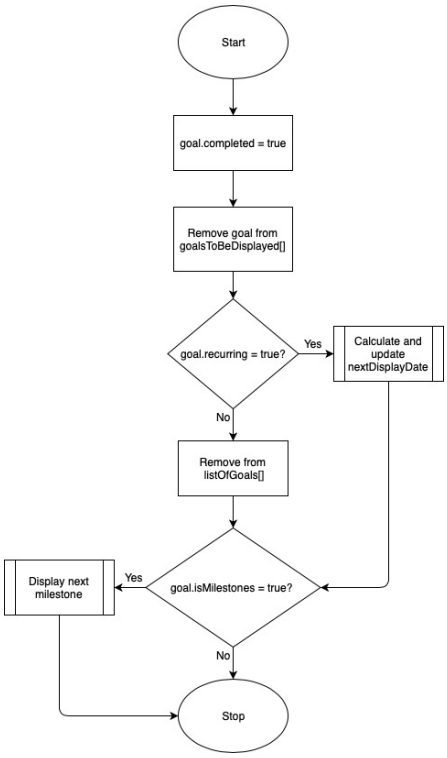


The sub-routine “Check for existing goals” will check the online database if the user has uploaded any previous goals. If they have, those goals will get pulled down by the application and stored in the listOfGoals array. A different function will then iterate through the list to check if any of the stored goals need to be displayed. If they do, then that goal will be copied to the goalsToBeDisplayed list. Another function will then be run to display any goals found in that list.

Completing a Goal

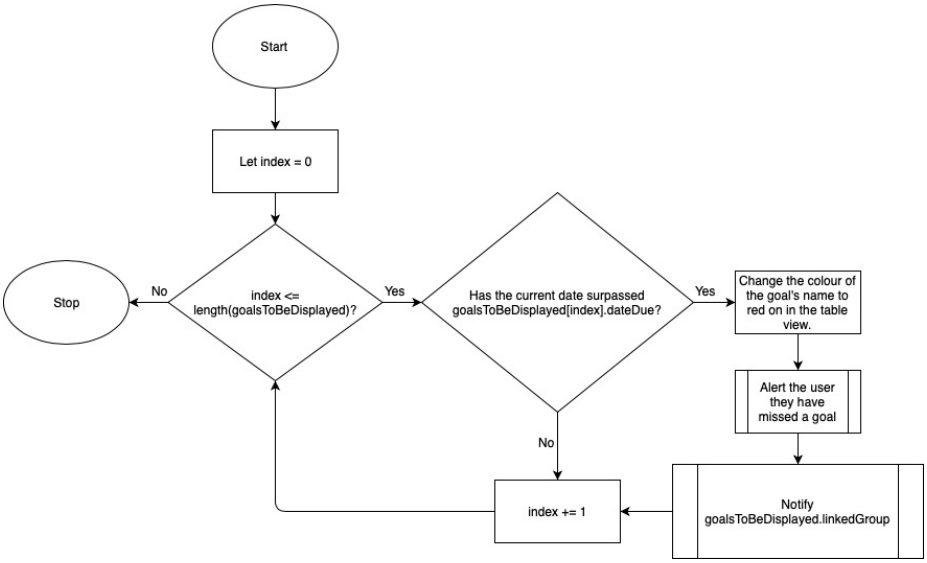
When the user completes a goal, the goal needs to be removed from goals to be displayed. The app then needs to check if the goal was simply a milestone to a greater project or if it has the recurring property. If it doesn’t, then the goal will be removed from the listOfGoals array.

Below is the flow chart for how this procedure will work:



Checking Whether a User Has Failed to Meet a Goal

The application also needs to check regularly whether a user has met a goal by the given deadline. Ideally, this function would run at the start of every day so the program can get real-time updates on a user's progress. However, the application needs to be open for this function to run. My solution is to run this function when the app is opened for the first time that day. The function will iterate through the goals and compare their due dates with the current date. If the due date has been and gone, the user will be notified, and so will the accountability group the goal is linked to.



This system does not automatically remove a goal the user has failed to meet and that is intentional. Just because the user has failed to meet a goal by the deadline they have set, it doesn't mean they don't still intend to complete it. Leaving it on the screen will hopefully encourage the user to pick themselves up and carry on regardless of deadlines.

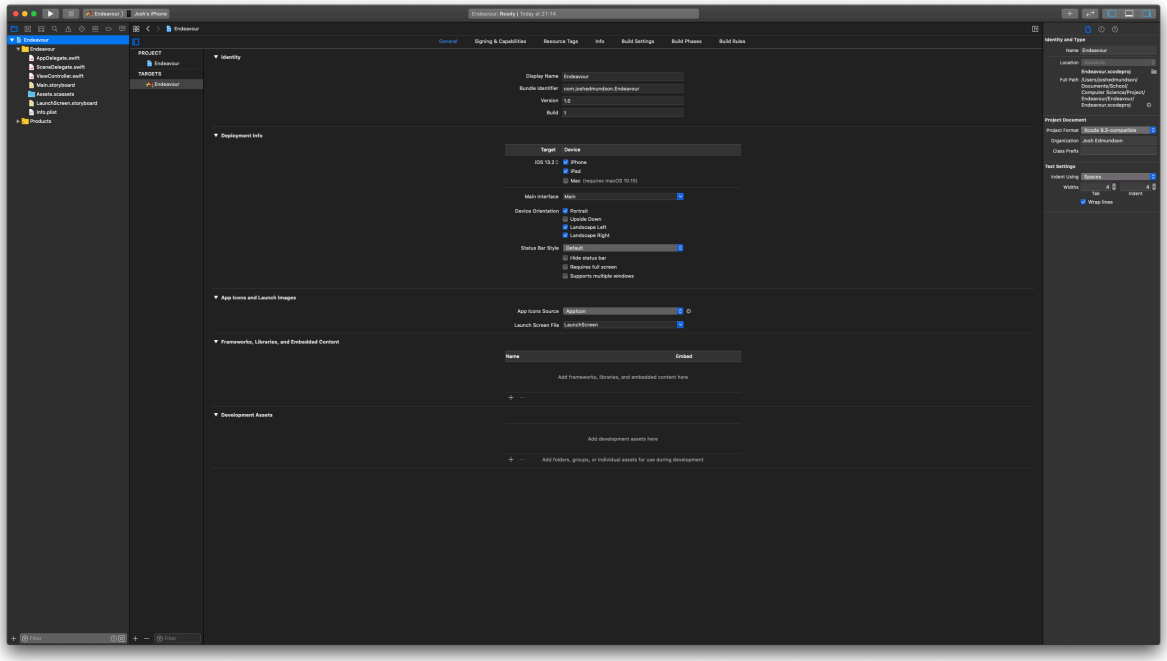
Additional Points

If the user pressure touches a goal on the app, they will be given the option to delete it.

Development

Setting up the initial project

Creating the file



This is the initial project as it appears in Xcode. I've set it up as a Swift file, targeting IOS 13.2. I'll make some general modifications to the devices this program is targeting because I don't want to have to worry about creating an app that works for both iPhone and iPad right from the start.

Removing the storyboard

When you use Xcode to create an IOS app, the standard way of building a GUI is through the use of storyboards. Storyboards allow you to construct a user interface by dragging and dropping components onto the screen. For this project, I am going to remove this feature so that I can build an interface programmatically, giving me a greater degree of control over the interface.

Step 1:

Delete the Main.storyboard file

Step 2:

Remove the code from in the Info.plist file and the project settings that tells the project to try and target Main.storyboard

Step 3:

Delete ViewController.swift and replace it with GoalTrackerScreen.swift

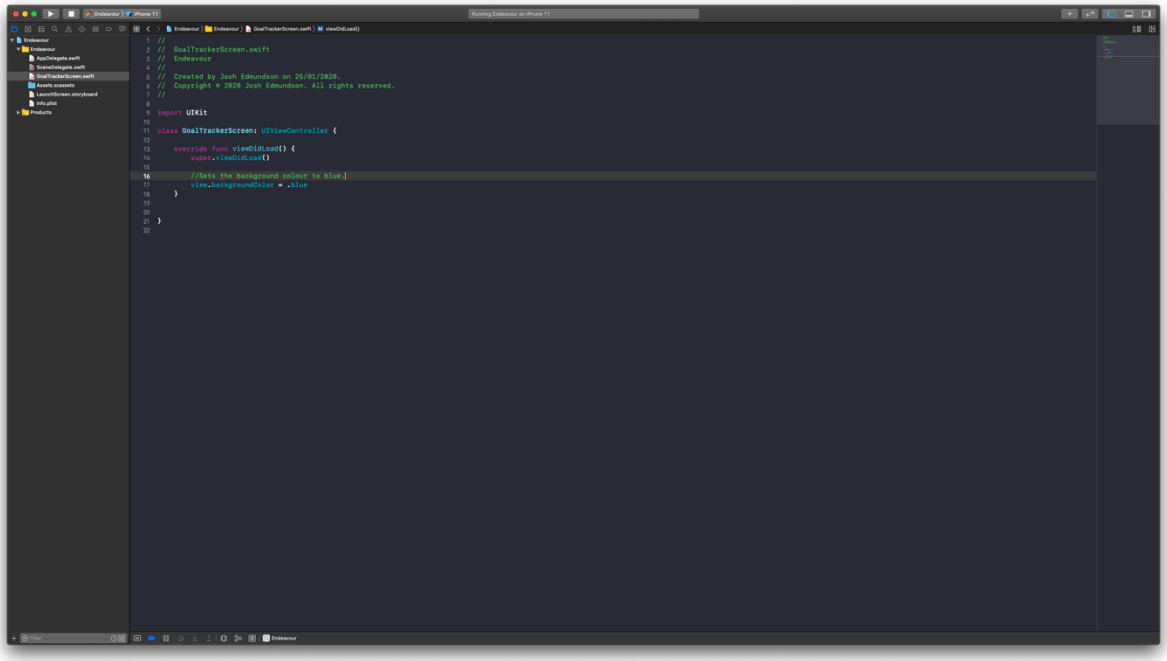
Step 4:

Create a new window and set it within GoalTrackerScreen as its root UIView by adding the code below to the SceneDelegate.swift file.

```
//This code creates a UIWindow object, sets its bounds to match the screen, places it within GoalTrackerScreen, and then makes it visible.
window = UIWindow(frame: windowScene.coordinateSpace.bounds)
window?.windowScene = windowScene
window?.rootViewController = GoalTrackerScreen()
window?.makeKeyAndVisible()
```

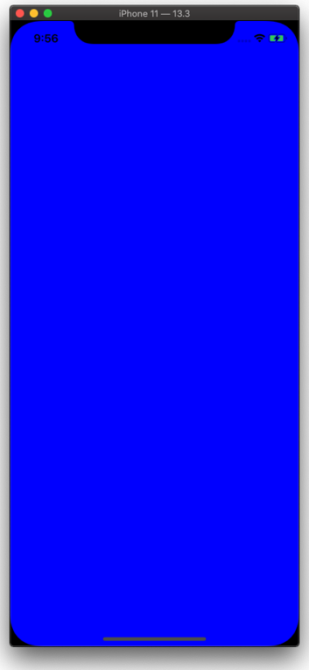
Step 5:

Check the transition from a storyboard based GUI to a programmatic one has been successful by changing the GoalTrackerScreen background to the colour blue programmatically.



Step 6:

Build and run the project.



Above is a screen shot of the iPhone simulator. As you can see, the screen is blue which shows the transition from a storyboard to a programmatic interface has been successful.

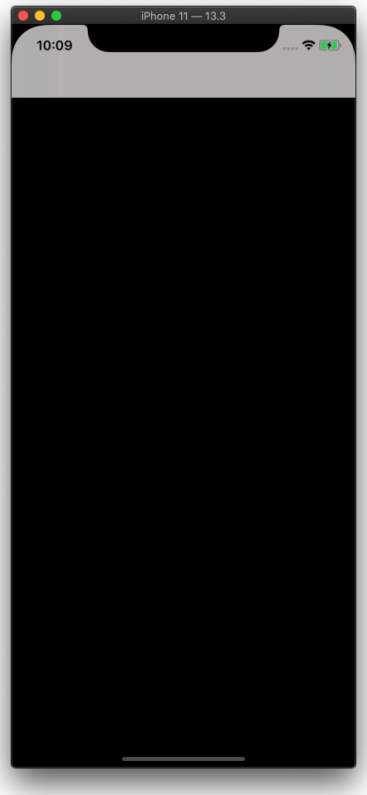
Creating a navigation controller

To allow the user to manoeuvre around the application, I need to create a navigation controller. This controller is where I will store navigation buttons that allow the user to move between different parts of the app.

```
//Create a GoalTrackerScreen object and a UINavigationController object. Place the UINavigationController within the GoalTrackerScreenObject
let goalTrackerScreen = GoalTrackerScreen()
let navigationController = UINavigationController(rootViewController: goalTrackerScreen)

//This code creates a UIWindow object, sets its bounds to match the screen, places it within the UINavigationController object, and then makes it visible.
window = UIWindow(frame: windowScene.coordinateSpace.bounds)
window?.windowScene = windowScene
window?.rootViewController = navigationController
window?.makeKeyAndVisible()
```

I've modified the code I previously added to the SceneDelegate.swift file to create a UINavigation object.

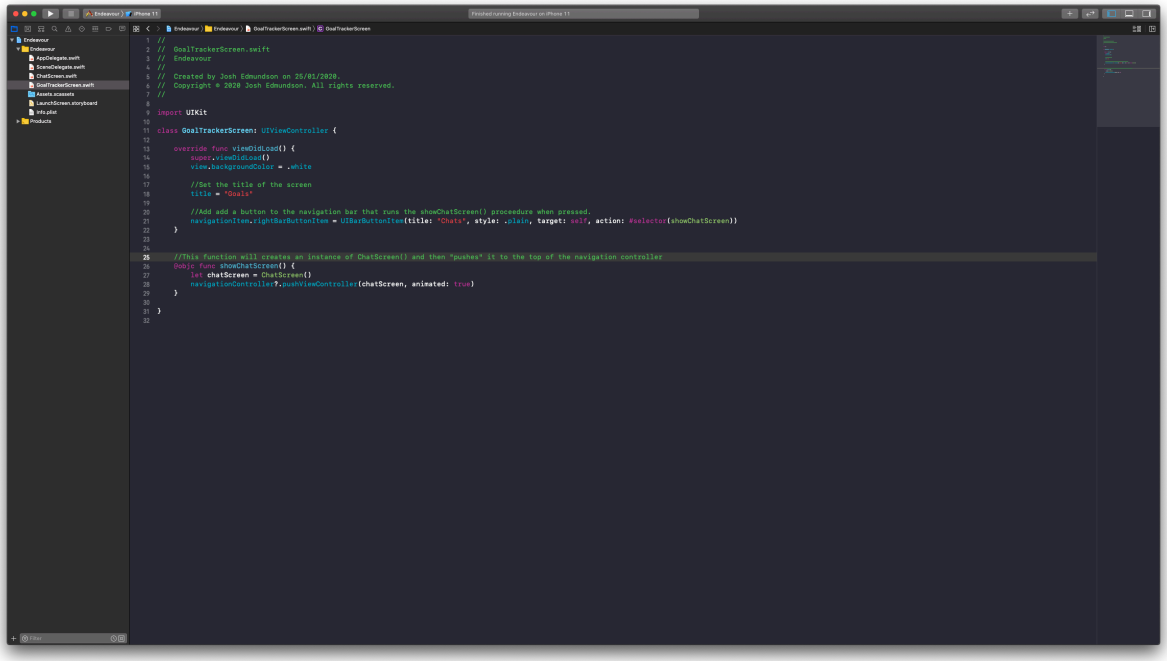
The idea behind this is to add a navigation bar to the top of our main screen, GoalTrackerScreen.



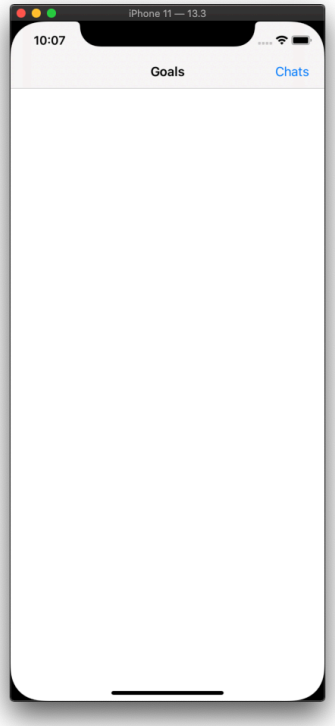
Now when I build the project and run it in the simulator, you can clearly see the navigation bar at the top of the screen. I have removed the code that made the screen turn blue.

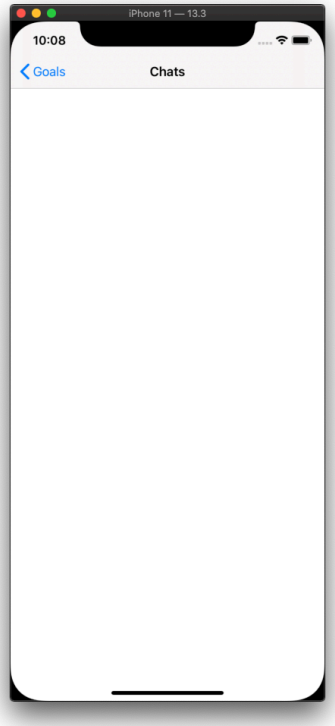
Navigating between screens

The next thing to do as part of the project setup is to create some way for the user to navigate between the Goal setting screen and the chat screen.



To do this, I added a button to the navigation bar I created previously that will “push” an instance of the ChatScreen() class to the top of the view when clicked.

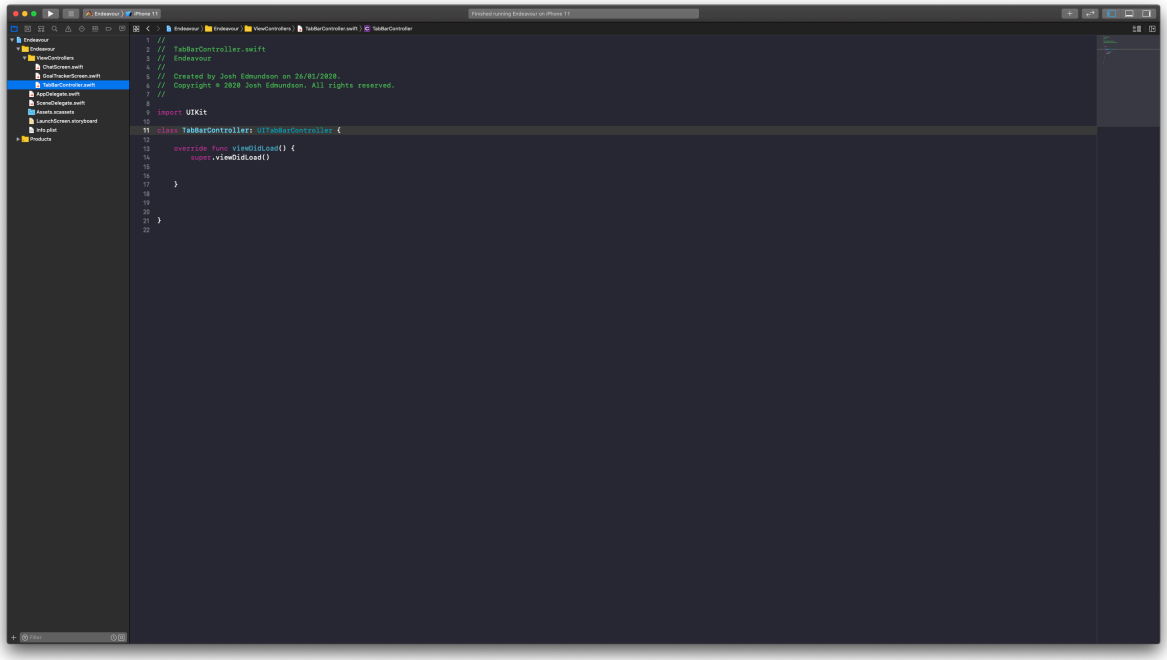




Replacing the navigation controller with a tab bar

After some though, I have decided that a tab bar at the bottom of the screen would be a more efficient way of switching between the GoalTrackerScreen and the ChatScreen as it would then allow me to have two individual navigation controllers for the two separate screens later on.

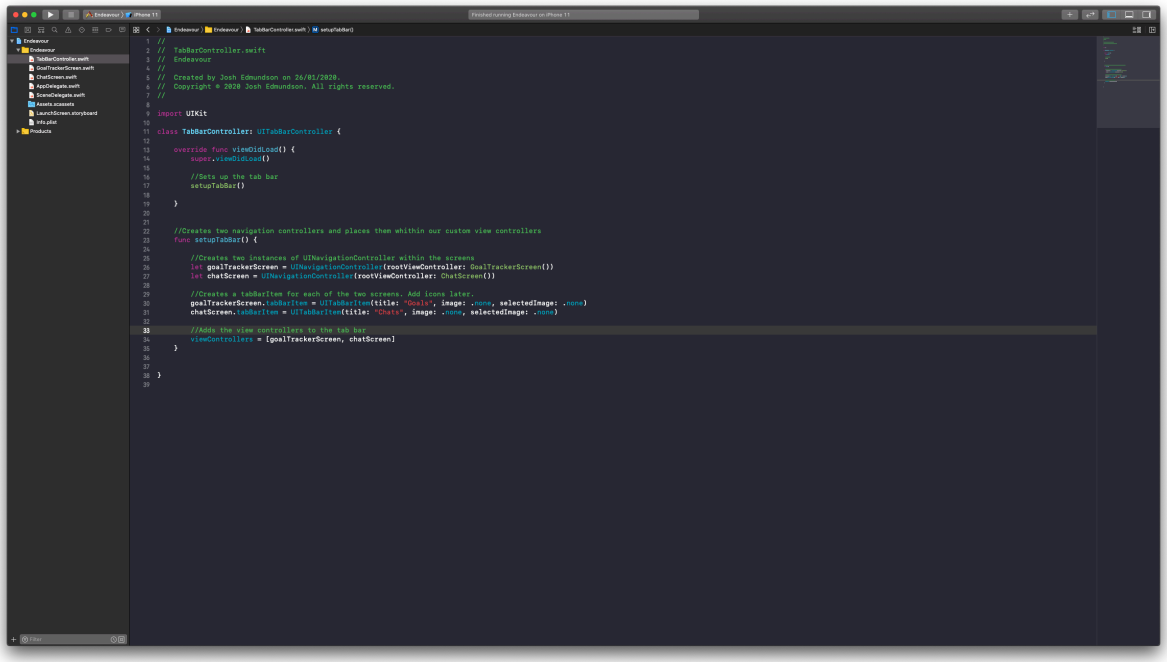
To replace the navigation controller, I have deleted the navigation controller code. I have also created a new class called TabBarController() which inherits from the UITabBarController() class.



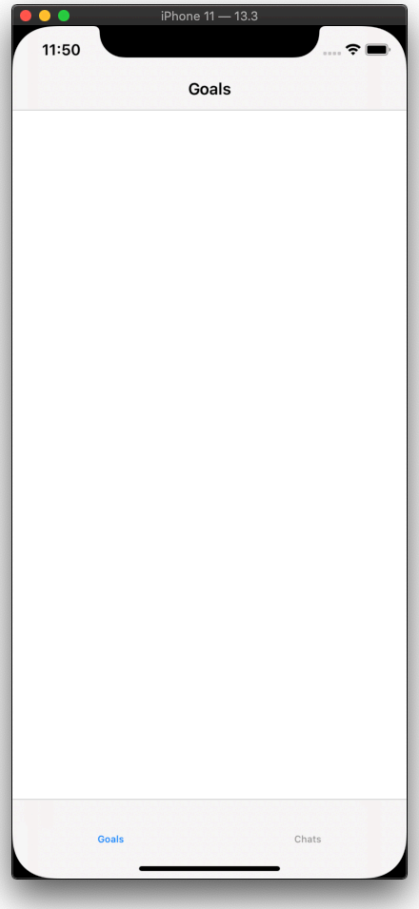
I have also altered the code in SceneDelegate.swift to make an instance of TabBarController the default view.

```
//Create an instance of TabBarController()
let tabBarController = TabBarController()

//This code creates a UIWindow object, sets its bounds to match the screen, places it within tabBarController, and then makes it visible.
window = UIWindow(frame: windowScene.coordinateSpace.bounds)
window?.windowScene = windowScene
window?.rootViewController = tabBarController
window?.makeKeyAndVisible()
```

I've added some code to the TabBarController class which allows the user to navigate between the two screens. I need to add accompanying icons to the tab bar at some point.



This is now what the app looks like:

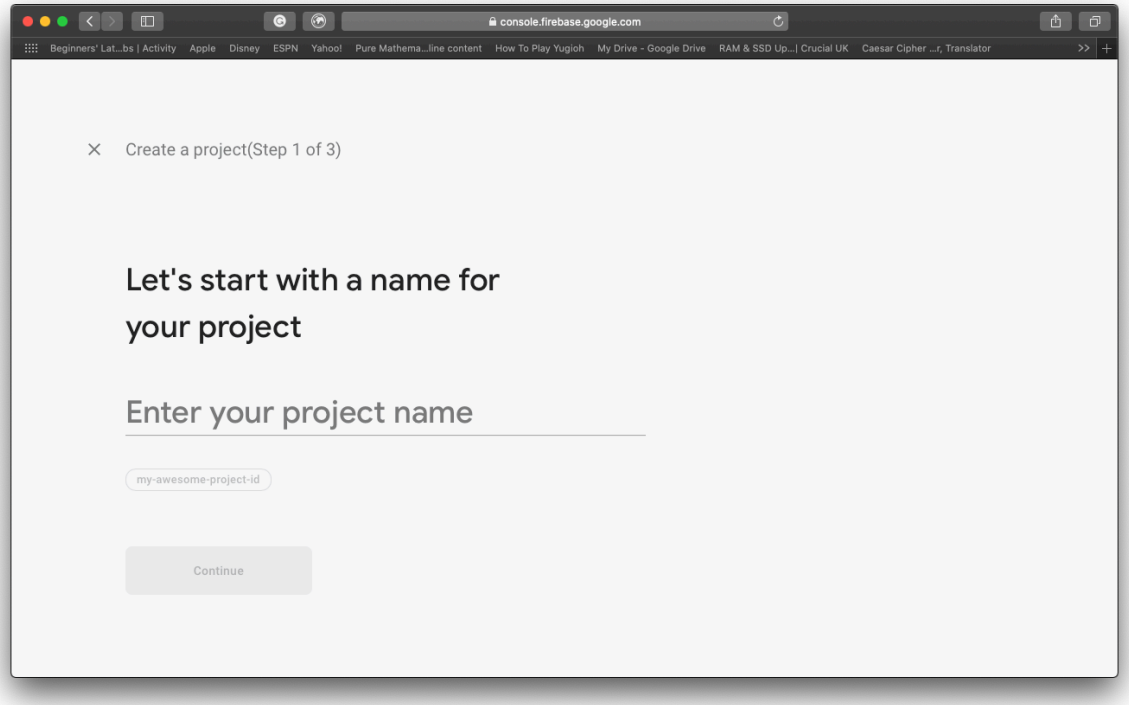


Adding Firebase

For this project to work, I'll need to use an online database to store data about the users, the groups they're in, and the goals they have.

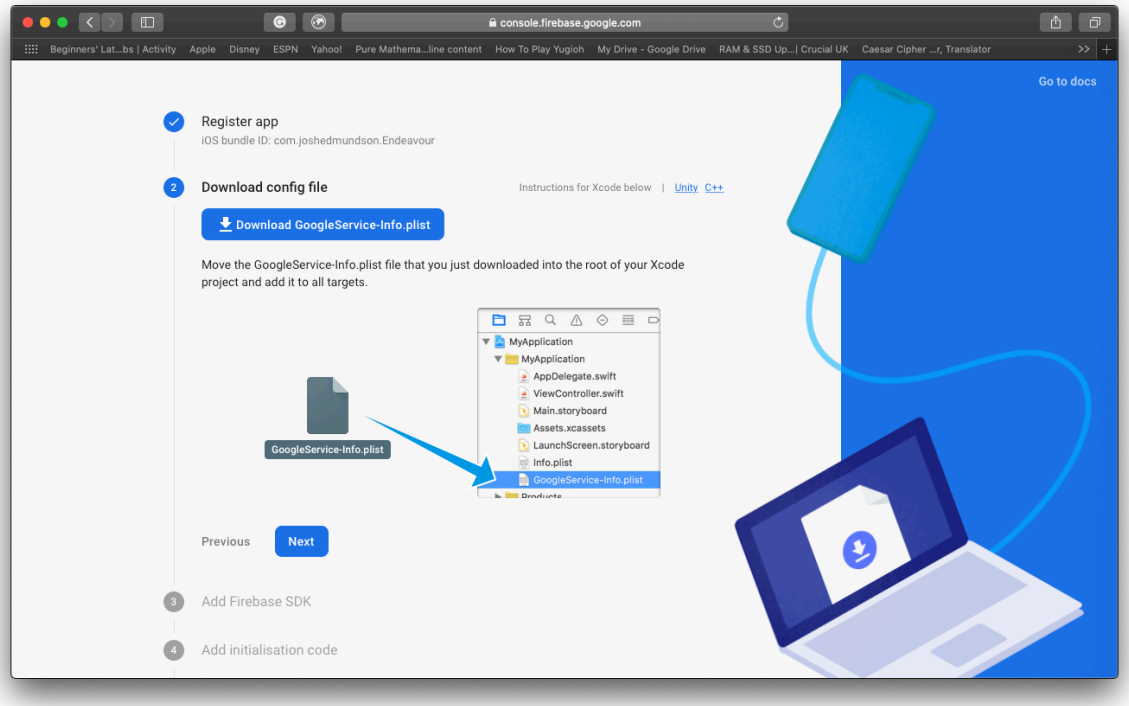
Step 1:

Create a new project on Firebase.



Step 2:

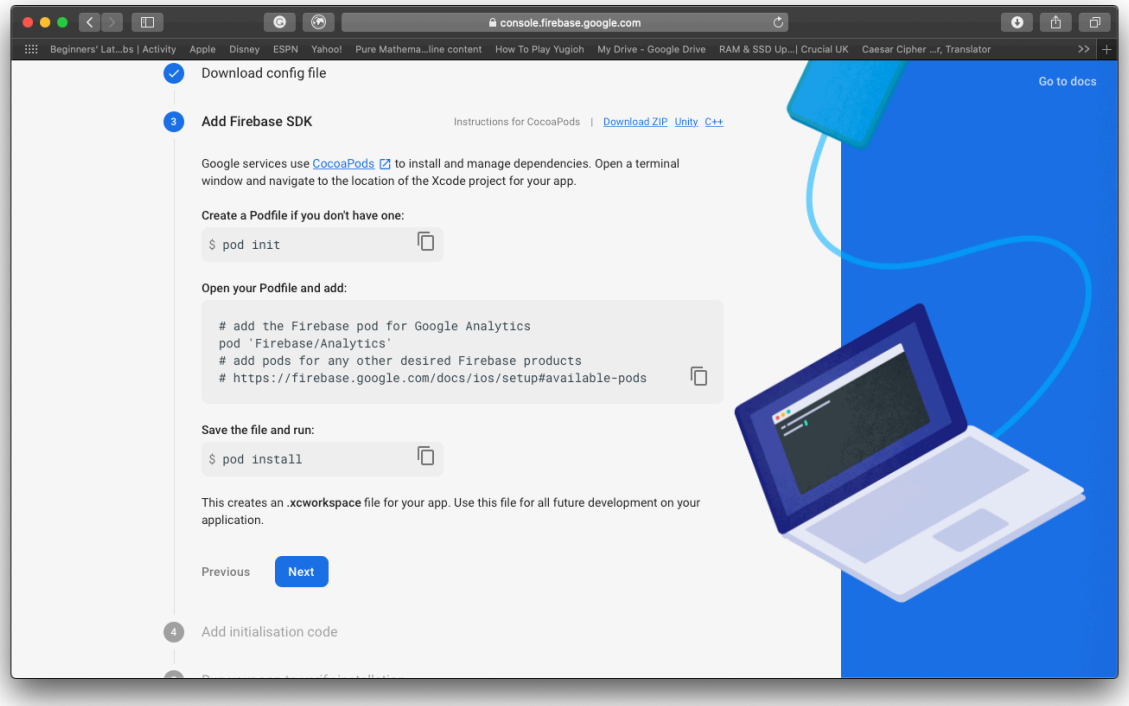
Add the new Firebase project to my app.



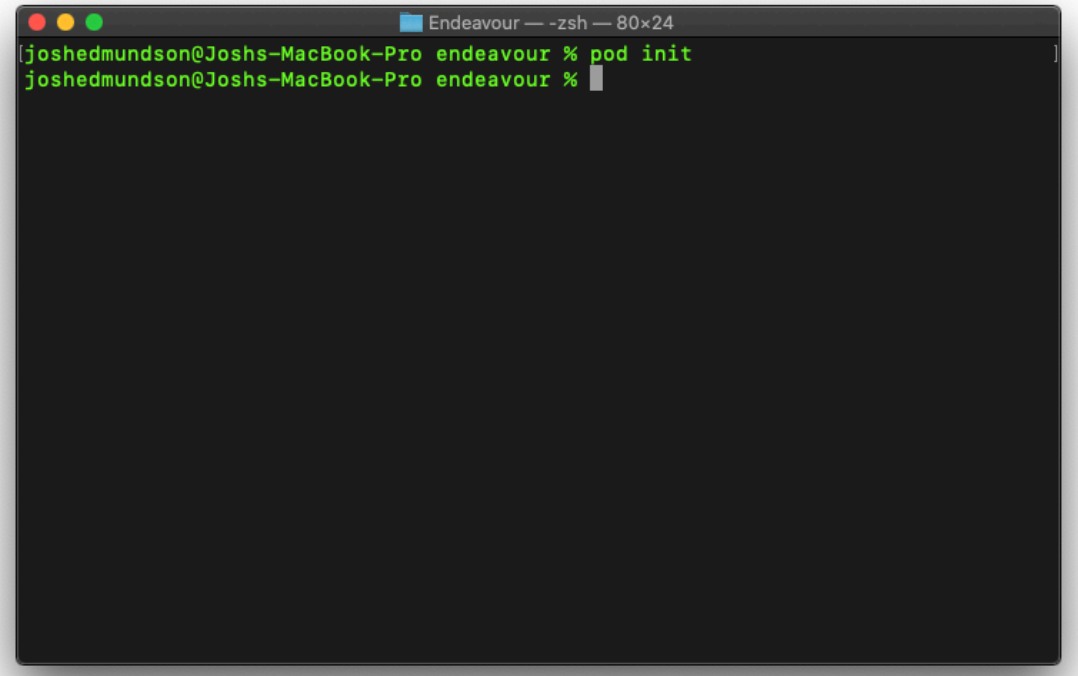
After giving my bundle identifier, I have to download a config file to connect my app to Firebase.

Having downloaded the config file and added it to my project, I now have to create a “pod” file and use a piece of software called Cocoa pods to download the relevant Firebase libraries to my project.

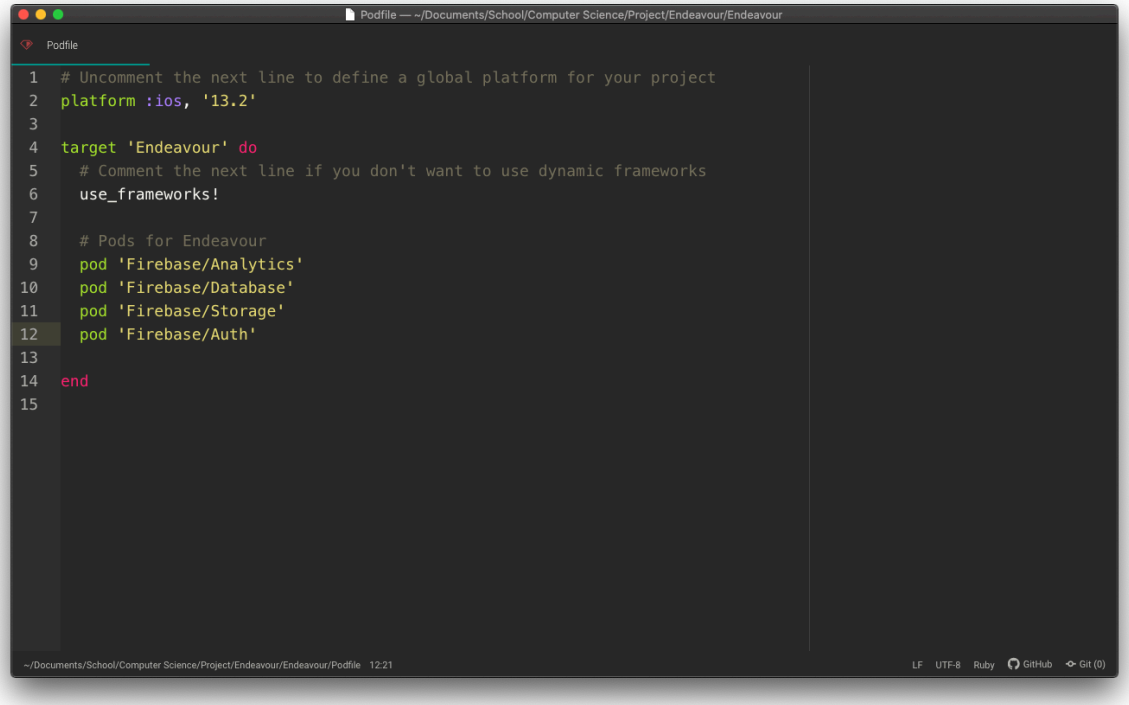
Step 3: Install pods



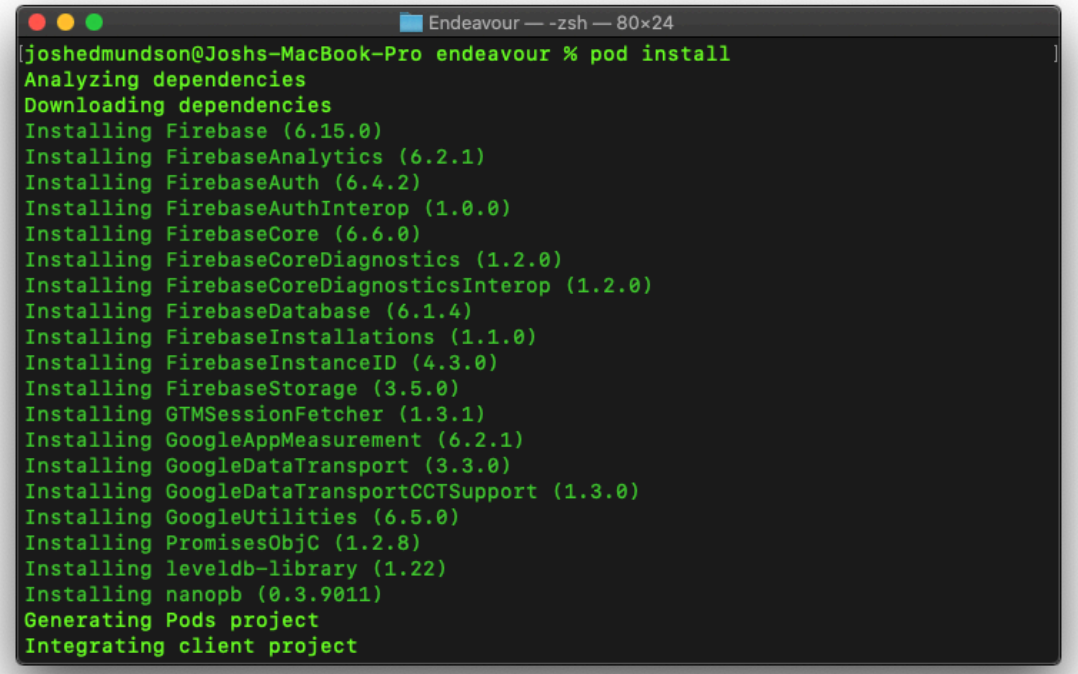
After opening a terminal in the “Endeavour” project director, I can now create a podfile.



I then add use the “pod” keyword to specify the libraries I want to add to my project.



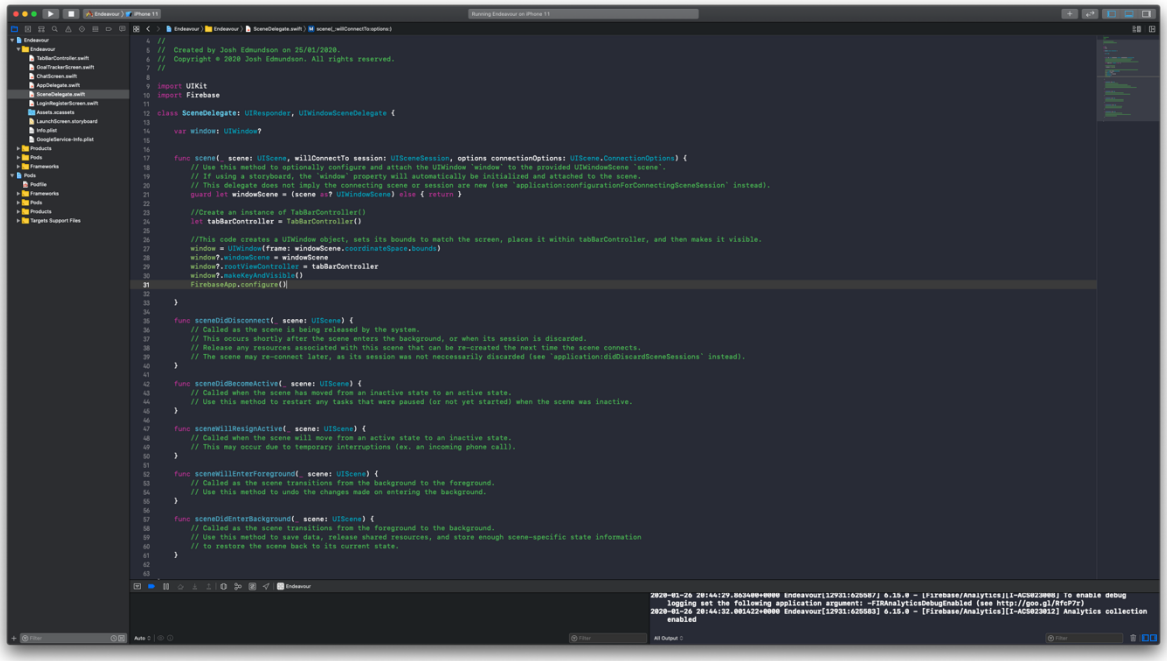
I can then run the pod install command to download the relevant files.



This has now created a .xcworkspace file which is essentially an Xcode project but with installed pods. I will carry on development in this file.

Step 4: Import and configure firebase

In the SceneDelegate.swift file I imported the Firebase library, and added Firebase.configure() to the scene function:



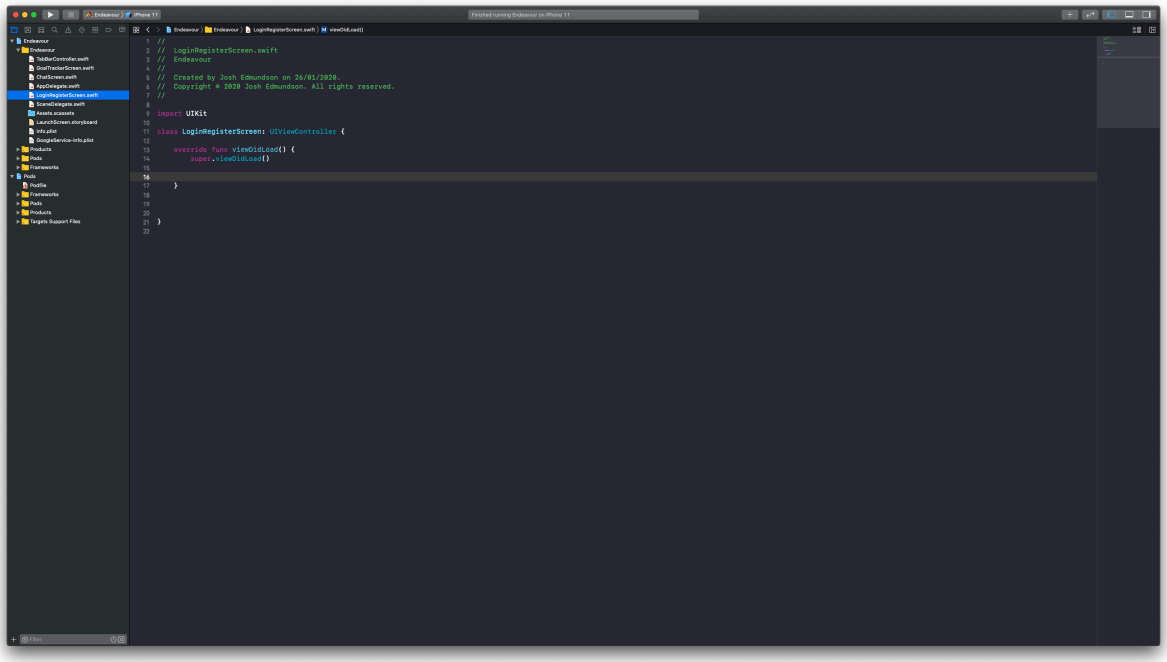
Firebase is now well and truly incorporated into the project.

Logging in

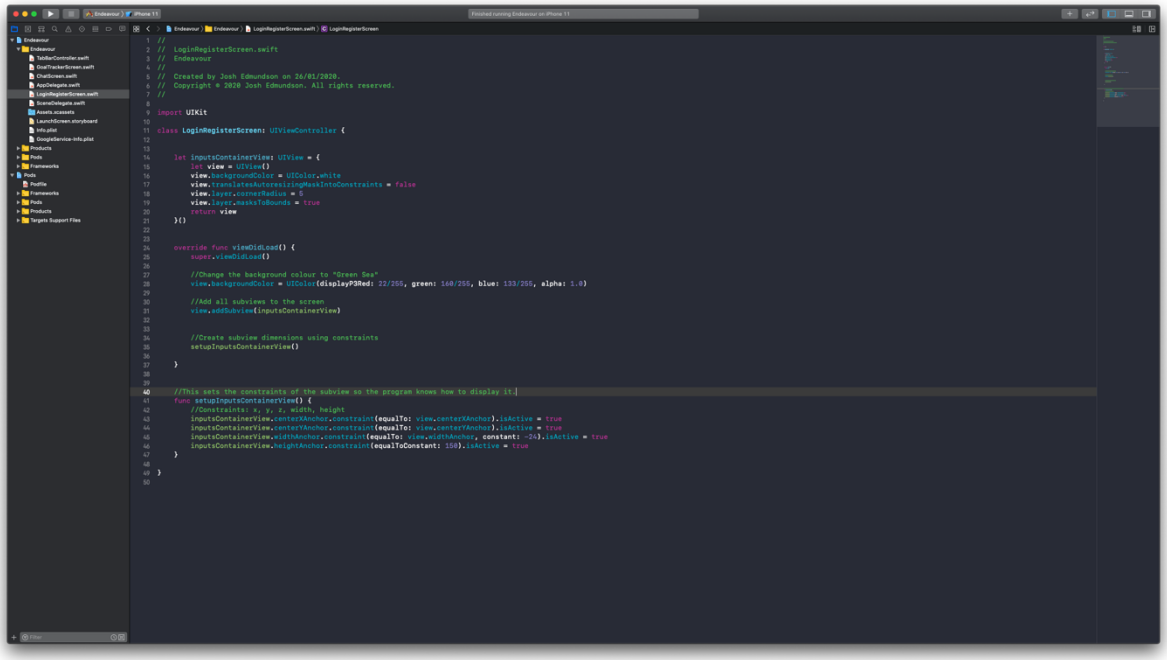
For this app, I plan to make full use of the incorporated Firebase database by using it to store the app users' information. This way, a user could log in to their account on any device.

Making the login page

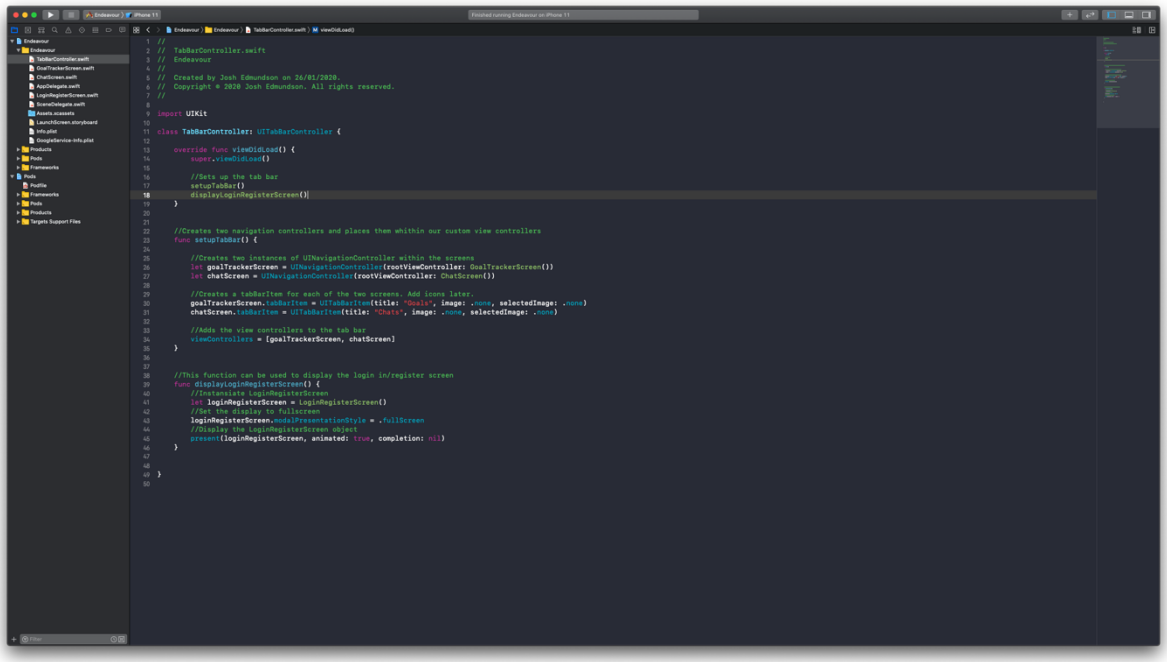
I've made a new class that inherits from UIViewController that will be the blueprint for the login screen. When the user loads up the app, they will be presented with the option to either log on with a pre-existing account or register with an email and password.



The first thing I will do is change the background colour of the login screen and then add a container that will hold all the text input fields where the user will be able to enter their information.

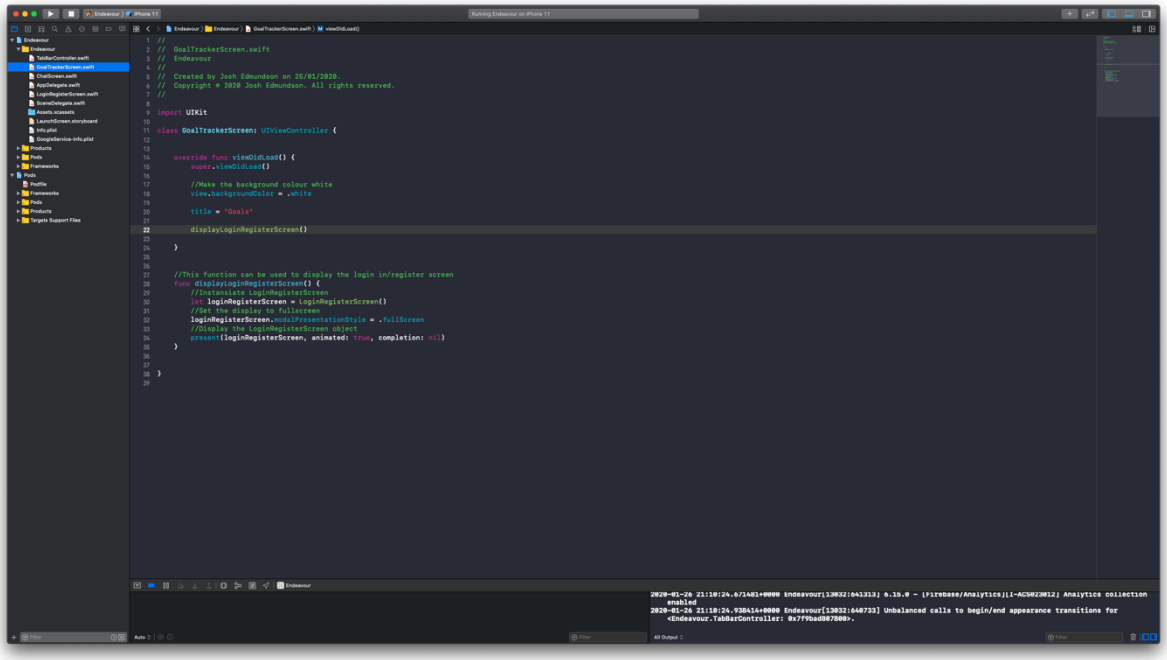


I want this view to be displayed whenever someone first runs the app so lets add that code to the TabBarController class.

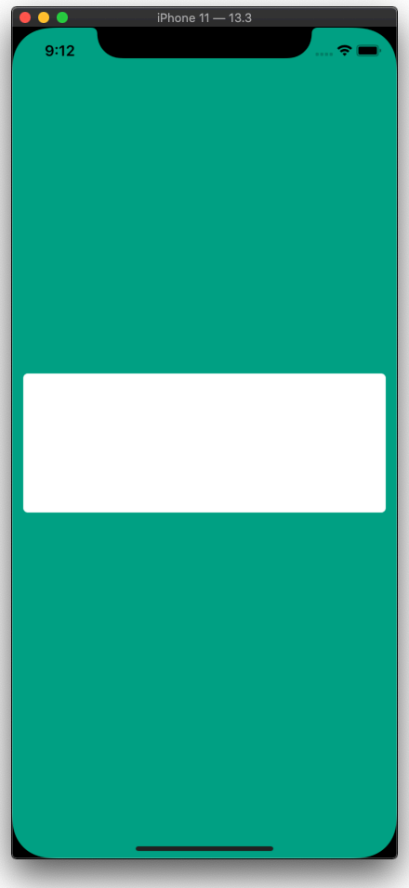


Here, the code I've added should make sure the login screen is displayed when the app is first run.

After building and running the program, the login register screen wasn't visible. In order to solve the problem, I moved the display function to the GoalTrackerScreen class instead, which worked.

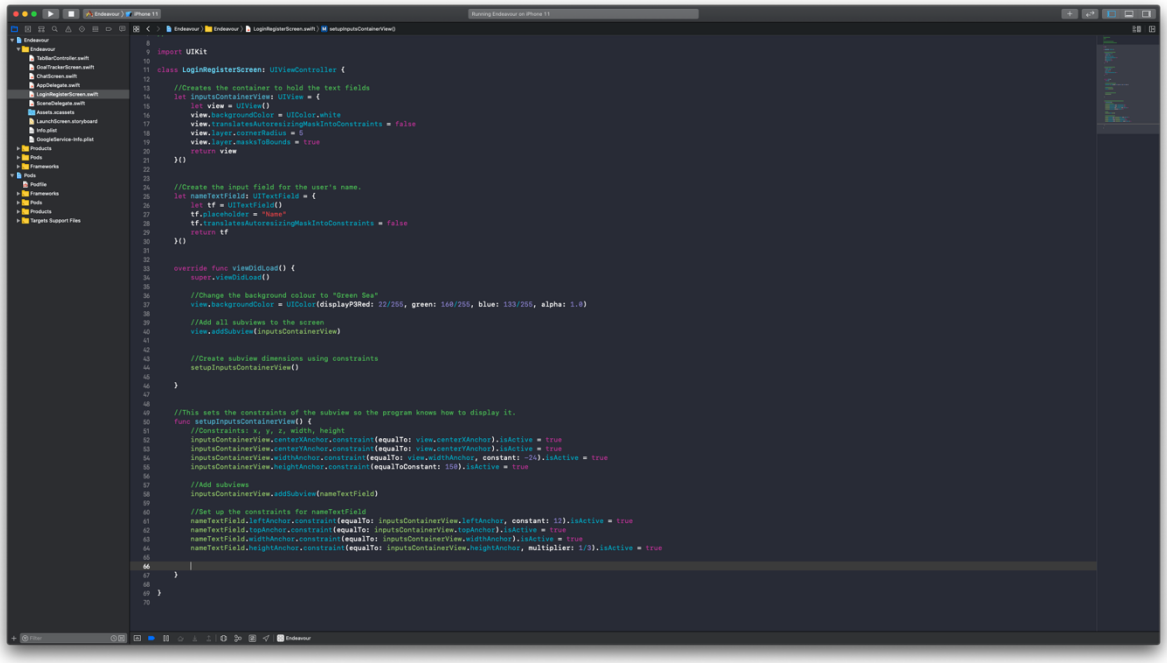


This is now what the app looks like when I run it:



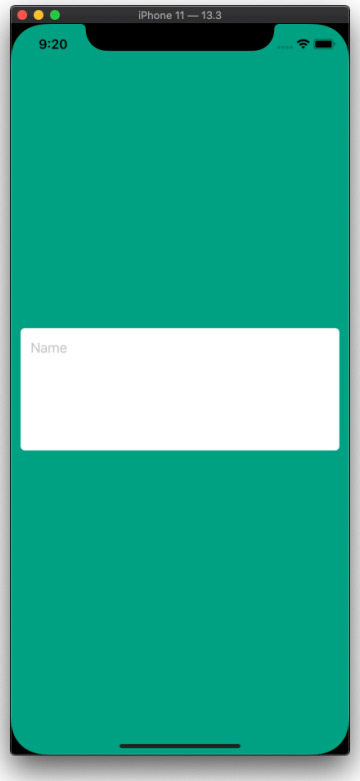
Now, I need to add in some actual input fields within this seemingly useless white box.

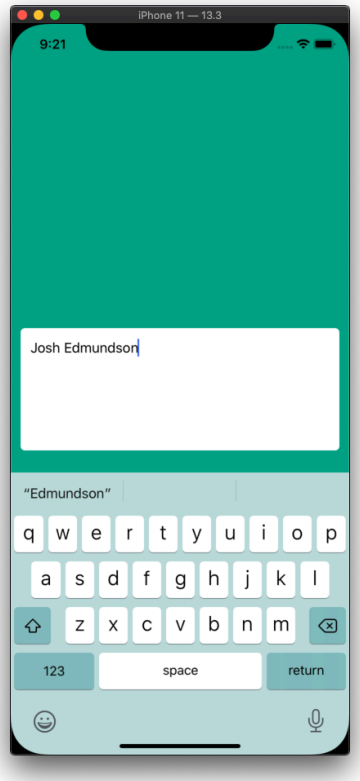
I will need to get three pieces of information from the user to create their account: their name, email, and password. The first input field I will make shall be for their name.



```
import UIKit

class LoginRegisterScreen: UIViewController {

    //Creates the container to hold the text fields
    let inputsContainerView: UIView = {
        let view = UIView()
        view.backgroundColor = UIColor.white
        view.translatesAutoresizingMaskIntoConstraints = false
        view.layer.cornerRadius = 5
        view.layer.masksToBounds = true
        return view
    }()


    //Create the input field for the user's name.
    let nameTextField: UITextField = {
        let tf = UITextField()
        tf.placeholder = "Name"
        tf.translatesAutoresizingMaskIntoConstraints = false
        return tf
    }()


    override func viewDidLoad() {
        super.viewDidLoad()

        //Change the background colour to "Green Sea"
        view.backgroundColor = UIColor(displayP3Red: 22/255, green: 160/255, blue: 133/255, alpha: 1.0)

        //Add all subviews to the screen
        view.addSubview(inputsContainerView)


        //Create subview dimensions using constraints
        setupInputsContainerView()

    }


    //This sets the constraints of the subview so the program knows how to display it.
    func setupInputsContainerView() {
        //Constraints: x, y, z, width, height
        inputsContainerView.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
        inputsContainerView.centerYAnchor.constraint(equalTo: view.centerYAnchor).isActive = true
        inputsContainerView.widthAnchor.constraint(equalTo: view.widthAnchor, constant: -24).isActive = true
        inputsContainerView.heightAnchor.constraint(equalToConstant: 150).isActive = true

        //Add subviews
        inputsContainerView.addSubview(nameTextField)

        //Set up the constraints for nameTextField
        nameTextField.leftAnchor.constraint(equalTo: inputsContainerView.leftAnchor, constant: 12).isActive = true
        nameTextField.topAnchor.constraint(equalTo: inputsContainerView.topAnchor).isActive = true
        nameTextField.widthAnchor.constraint(equalTo: inputsContainerView.widthAnchor).isActive = true
        nameTextField.heightAnchor.constraint(equalTo: inputsContainerView.heightAnchor, multiplier: 1/3).isActive = true

    }
}
```

What I've done here is create an instance of the class UITextField called nameTextField, modified its attributes, and then added it as a subview of inputsContainerView. This will place the input field within the white box you saw on screen.

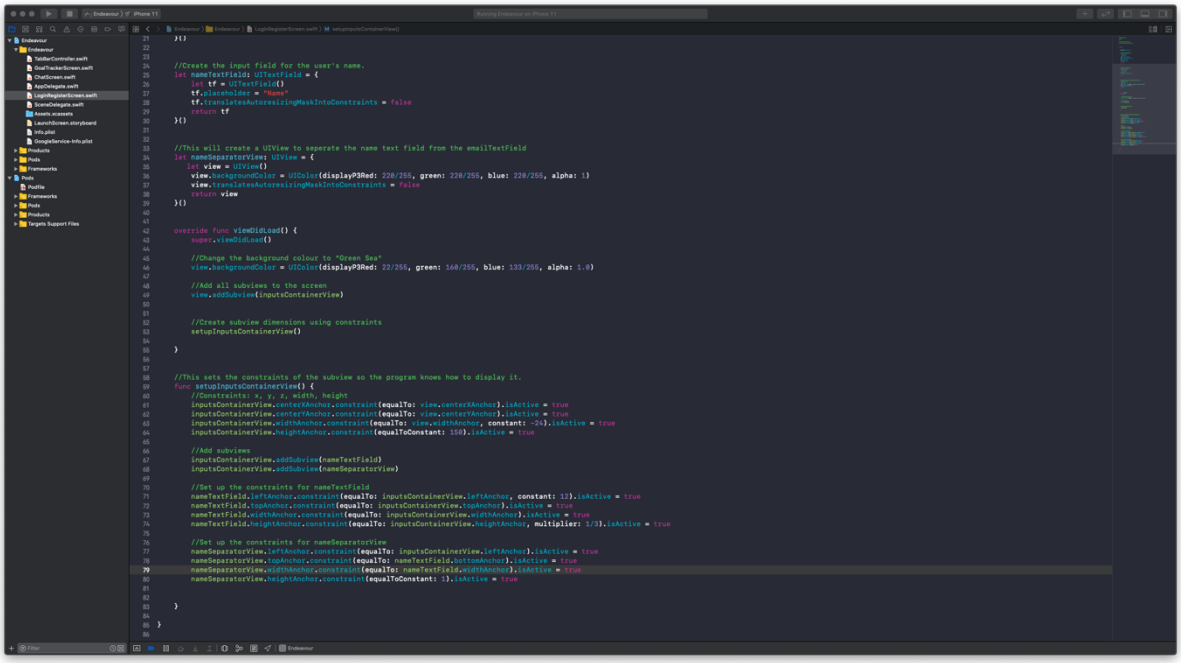




Now we have a name input field.

When you add views to a view controller in swift, you can create a dynamic display by placing views relative to each other. This is what anchors are used for.

To separate the various text fields, I've created a new UIView called nameSeparatorView that will underline the top two text boxes to give them definite boundaries.



```
}()


//Create the input field for the user's name.
let nameTextField: UITextField = {
    let tf = UITextField()
    tf.placeholder = "Name"
    tf.translatesAutoresizingMaskIntoConstraints = false
    return tf
}()


//This will create a UIView to seperate the name text field from the emailTextField
let nameSeparatorView: UIView = {
    let view = UIView()
    view.backgroundColor = UIColor(displayP3Red: 220/255, green: 220/255, blue: 220/255, alpha: 1)
    view.translatesAutoresizingMaskIntoConstraints = false
    return view
}()


override func viewDidLoad() {
    super.viewDidLoad()

    //Change the background colour to "Green Sea"
    view.backgroundColor = UIColor(displayP3Red: 22/256, green: 160/255, blue: 133/255, alpha: 1.0)

    //Add all subviews to the screen
    view.addSubview(inputsContainerView)


    //Create subview dimensions using constraints
    setupInputsContainerView()

}


//This sets the constraints of the subview so the program knows how to display it.
func setupInputsContainerView() {
    //Constraints: x, y, z, width, height
    inputsContainerView.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
    inputsContainerView.centerYAnchor.constraint(equalTo: view.centerYAnchor).isActive = true
    inputsContainerView.widthAnchor.constraint(equalTo: view.widthAnchor, constant: -24).isActive = true
    inputsContainerView.heightAnchor.constraint(equalToConstant: 150).isActive = true

    //Add subviews
    inputsContainerView.addSubview(nameTextField)
    inputsContainerView.addSubview(nameSeparatorView)

    //Set up the constraints for nameTextField
    nameTextField.leftAnchor.constraint(equalTo: inputsContainerView.leftAnchor, constant: 12).isActive = true
    nameTextField.topAnchor.constraint(equalTo: inputsContainerView.topAnchor).isActive = true
    nameTextField.widthAnchor.constraint(equalTo: inputsContainerView.widthAnchor).isActive = true
    nameTextField.heightAnchor.constraint(equalTo: inputsContainerView.heightAnchor, multiplier: 1/3).isActive = true

    //Set up the constraints for nameSeparatorView
    nameSeparatorView.leftAnchor.constraint(equalTo: inputsContainerView.leftAnchor).isActive = true
    nameSeparatorView.topAnchor.constraint(equalTo: nameTextField.bottomAnchor).isActive = true
    nameSeparatorView.widthAnchor.constraint(equalTo: nameTextField.widthAnchor).isActive = true
    nameSeparatorView.heightAnchor.constraint(equalToConstant: 1).isActive = true


}
}
```

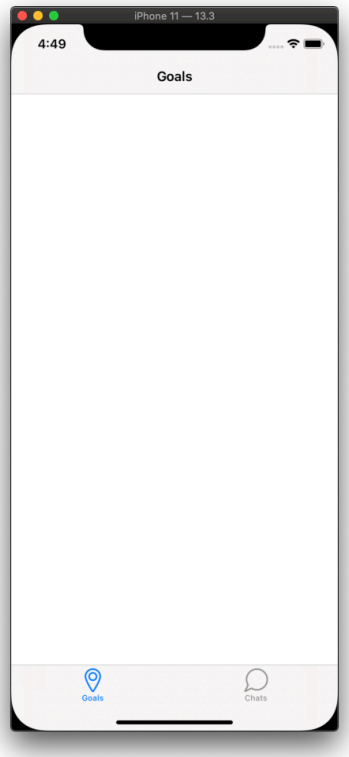
The simulation now looks like this:

Now, I need to make two more input text fields, one for the user's email, the other for their password.

I have also managed to find some suitable icons and have added them to the tab bar:

```
goalTrackerScreen.tabBarItem.title = "Goals"
goalTrackerScreen.tabBarItem.image = UIImage(named: "goals_pin_unselected")
goalTrackerScreen.tabBarItem.selectedImage = UIImage(named: "goals_pin_selected")

chatScreen.tabBarItem.title = "Chats"
chatScreen.tabBarItem.image = UIImage(named: "messages_unselected")
chatScreen.tabBarItem.selectedImage = UIImage(named: "messages_selected")
```

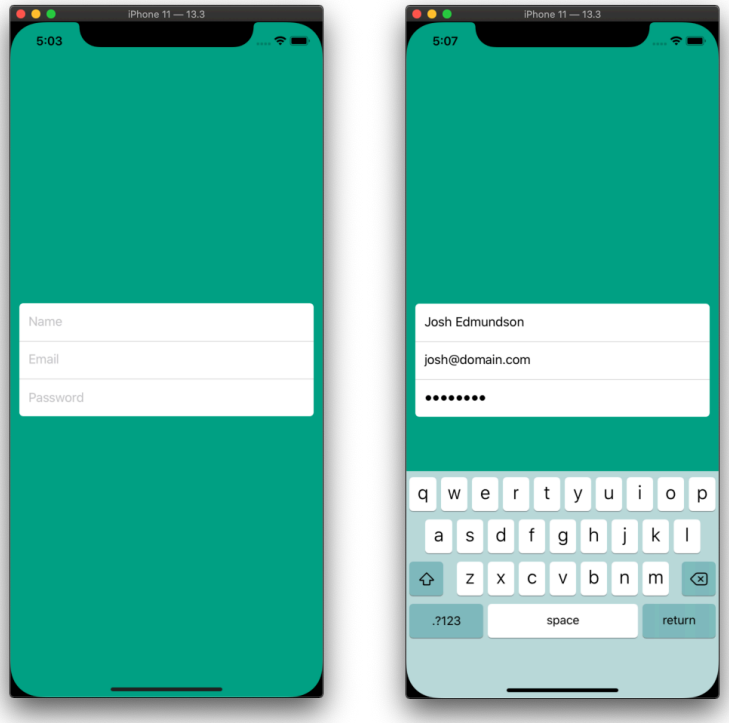
So now our instance of TabBarViewController() looks like this:



I have now added the final two text input fields, one for the user's email and the other for their password. Below is the whole code for the LoginRegisterScreen class:

```swift
//
//  LoginRegisterScreen.swift
//  Endeavour
//
//  Created by Josh Edmundson on 26/01/2020.
//  Copyright © 2020 Josh Edmundson. All rights reserved.
//

import UIKit

class LoginRegisterScreen: UIViewController {

    //Creates the container to hold the text fields
    let inputsContainerView: UIView = {
        let view = UIView()
        view.backgroundColor = UIColor.white
        view.translatesAutoresizingMaskIntoConstraints = false
        view.layer.cornerRadius = 5
        view.layer.masksToBounds = true
        return view
    }()


    //Create the input field for the user's name.
    let nameTextField: UITextField = {
        let tf = UITextField()
        tf.placeholder = "Name"
        tf.translatesAutoresizingMaskIntoConstraints = false
        return tf
    }()


    //This will create a UIView to seperate the name text field from the emailTextField
    let nameSeparatorView: UIView = {
       let view = UIView()
        view.backgroundColor = UIColor(displayP3Red: 220/255, green: 220/255, blue: 220/255, alpha: 1)
        view.translatesAutoresizingMaskIntoConstraints = false
        return view
    }()


    //Create email text field
    let emailTextField: UITextField = {
        let tf = UITextField()
        tf.placeholder = "Email"
        tf.translatesAutoresizingMaskIntoConstraints = false
        return tf
    }()

    //Create the email and password text field seperator
    let emailSeparatorView: UIView = {
       let view = UIView()
        view.backgroundColor = UIColor(displayP3Red: 220/255, green: 220/255, blue: 220/255, alpha: 1)
        view.translatesAutoresizingMaskIntoConstraints = false
        return view
    }()



    //Creates the password text field
    let passwordTextField: UITextField = {
        let tf = UITextField()
        tf.placeholder = "Password"
        tf.translatesAutoresizingMaskIntoConstraints = false
        tf.isSecureTextEntry = true //This will hide the user input.
        return tf
    }()


    override func viewDidLoad() {
        super.viewDidLoad()

        //Change the background colour to "Green Sea"
        view.backgroundColor = UIColor(displayP3Red: 22/255, green: 160/255, blue: 133/255, alpha: 1.0)

        //Add all subviews to the screen
        view.addSubview(inputsContainerView)


        //Create subview dimensions using constraints
        setupInputsContainerView()

    }


    //This sets the constraints of the subview so the program knows how to display it.
    func setupInputsContainerView() {
        //Constraints: x, y, z, width, height
        inputsContainerView.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
        inputsContainerView.centerYAnchor.constraint(equalTo: view.centerYAnchor).isActive = true
        inputsContainerView.widthAnchor.constraint(equalTo: view.widthAnchor, constant: -24).isActive = true
        inputsContainerView.heightAnchor.constraint(equalToConstant: 150).isActive = true


        //Add subviews
        inputsContainerView.addSubview(nameTextField)
        inputsContainerView.addSubview(nameSeparatorView)
        inputsContainerView.addSubview(emailTextField)
        inputsContainerView.addSubview(emailSeparatorView)
        inputsContainerView.addSubview(passwordTextField)


        //Set up the constraints for nameTextField
        nameTextField.leftAnchor.constraint(equalTo: inputsContainerView.leftAnchor, constant: 12).isActive = true
        nameTextField.topAnchor.constraint(equalTo: inputsContainerView.topAnchor).isActive = true
        nameTextField.widthAnchor.constraint(equalTo: inputsContainerView.widthAnchor).isActive = true
        nameTextField.heightAnchor.constraint(equalTo: inputsContainerView.heightAnchor, multiplier: 1/3).isActive = true

        //Set up the constraints for nameSeparatorView
        nameSeparatorView.leftAnchor.constraint(equalTo: inputsContainerView.leftAnchor).isActive = true
        nameSeparatorView.topAnchor.constraint(equalTo: nameTextField.bottomAnchor).isActive = true
        nameSeparatorView.widthAnchor.constraint(equalTo: nameTextField.widthAnchor).isActive = true
        nameSeparatorView.heightAnchor.constraint(equalToConstant: 1).isActive = true


        //Set up the constraints for emailTextField
        emailTextField.leftAnchor.constraint(equalTo: inputsContainerView.leftAnchor, constant: 12).isActive = true
        emailTextField.topAnchor.constraint(equalTo: nameTextField.bottomAnchor).isActive = true
        emailTextField.widthAnchor.constraint(equalTo: inputsContainerView.widthAnchor).isActive = true
        emailTextField.heightAnchor.constraint(equalTo: inputsContainerView.heightAnchor, multiplier: 1/3).isActive = true

        //Set up constraints for emailSeparatorView
        emailSeparatorView.leftAnchor.constraint(equalTo: inputsContainerView.leftAnchor).isActive = true
        emailSeparatorView.topAnchor.constraint(equalTo: emailTextField.bottomAnchor).isActive = true
        emailSeparatorView.widthAnchor.constraint(equalTo: nameTextField.widthAnchor).isActive = true
        emailSeparatorView.heightAnchor.constraint(equalToConstant: 1).isActive = true


        //Set up constraints for passwordTextField
        passwordTextField.leftAnchor.constraint(equalTo: inputsContainerView.leftAnchor, constant: 12).isActive = true
        passwordTextField.topAnchor.constraint(equalTo: emailSeparatorView.bottomAnchor).isActive = true
        passwordTextField.widthAnchor.constraint(equalTo: inputsContainerView.widthAnchor).isActive = true
        passwordTextField.heightAnchor.constraint(equalTo: inputsContainerView.heightAnchor, multiplier: 1/3).isActive = true

    }
}

```

Here is now what the app looks like in the simulator:



Now, the user could really do with some way of submitting there information, so I will add a login/register button beneath the input text fields.

This defines the attributes of the button:

```
//Create the login/register button
let loginRegisterButton: UIButton = {
    let button = UIButton(type: .system)
    button.backgroundColor = UIColor(displayP3Red: 80/255, green: 101/255, blue: 161/255, alpha: 1)
    button.setTitle("Register", for: .normal)
    button.setTitleColor(UIColor.white, for: .normal)
    //Always set this to false
    button.translatesAutoresizingMaskIntoConstraints = false
    button.titleLabel?.font = UIFont.boldSystemFont(ofSize: 16)

    //Makes the button run the handleRegisterFunction when the user releases the button within its boundary
    button.addTarget(self, action: #selector(handleRegister), for: .touchUpInside)

    return button
}()
```

Note the error: “Use of unresolved identifier ‘handleRegister’”. The reason I am getting this error is because I am telling the button to run the function handleRegister() when it is pressed but I have not actually made it yet. Let’s resolve that issue:

```
@objc func handleRegister() {
    print(123)
}
```

For the time being, this function is simply going to print out ‘123’ to the console when called. After I have finished setting up the button on the screen, I will return to this function.

```
//Function to setup all the constraints neccessary to give loginRegisterButton dimensions and a position on the screen
func setupLoginRegisterButton() {
    loginRegisterButton.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
    loginRegisterButton.topAnchor.constraint(equalTo: inputsContainerView.bottomAnchor, constant: 12).isActive = true
    loginRegisterButton.widthAnchor.constraint(equalTo: inputsContainerView.widthAnchor).isActive = true
    loginRegisterButton.heightAnchor.constraint(equalToConstant: 50).isActive = true
}
```

Above sets up the constraints within a function which I then call in the viewDidLoad() method.

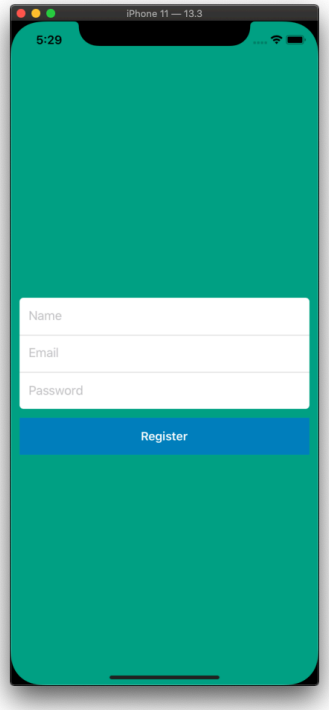
When I tried to build and run the project, I got this error:

```
2020-01-27 17:18:52.056824+0000 Endeavour[5293:188991] *** Terminating app due to uncaught exception
    'NSGenericException', reason: 'Unable to activate constraint with anchors <NSLayoutXAxisAnchor:0x6000015bdf80
    "UIButton:0x7fba2b519310'Register'.centerX"> and <NSLayoutXAxisAnchor:0x6000015bd440
    "UIView:0x7fba2b621c50.centerX"> because they have no common ancestor.  Does the constraint or its anchors reference
    items in different view hierarchies?  That's illegal.'
*** First throw call stack:
(
    0   CoreFoundation                      0x000000010cb1d27e __exceptionPreprocess + 350
    1   libobjc.A.dylib                     0x000000010c98ab20 objc_exception_throw + 48
    2   Foundation                          0x000000010bbe0438 -[NSLayoutConstraint setActive:] + 0
    3   Endeavour                           0x000000010a72038b $s9Endeavour19LoginRegisterScreenC05setupbC6ButtonyyF +
        459
    4   Endeavour                           0x000000010a71e160 $s9Endeavour19LoginRegisterScreenC11viewDidLoadyyF + 800
    5   Endeavour                           0x000000010a71e23b $s9Endeavour19LoginRegisterScreenC11viewDidLoadyyFTo + 43
    6   UIKitCore                           0x00000001108c7f01 -[UIViewController
        _sendViewDidLoadWithAppearanceProxyObjectTaggingEnabled] + 83
    7   UIKitCore                           0x00000001108cce5a -[UIViewController loadViewIfRequired] + 1084
    8   UIKitCore                           0x00000001108cd277 -[UIViewController view] + 27
    9   UIKitCore                           0x00000001107e540b -[_UIFullscreenPresentationController
        _setPresentedViewController:] + 84
    10  UIKitCore                           0x00000001107d85c1 -[UIPresentationController
        initWithPresentedViewController:presentingViewController:] + 184
    11  UIKitCore                           0x00000001108de450 -[UIViewController
        _presentViewController:withAnimationController:completion:] + 3509
    12  UIKitCore                           0x00000001108e101b __63-[UIViewController
        _presentViewController:animated:completion:]_block_invoke + 98
    13  UIKitCore                           0x00000001108e1533 -[UIViewController
        _performCoordinatedPresentOrDismiss:animated:] + 511
    14  UIKitCore                           0x00000001108e0f79 -[UIViewController
        _presentViewController:animated:completion:] + 187
    15  UIKitCore                           0x00000001108e11e0 -[UIViewController
        presentViewController:animated:completion:] + 150
    16  Endeavour                           0x000000010a7232bf
        $s9Endeavour17GoalTrackerScreenC020displayLoginRegisterD0yyF + 143
    17  Endeavour                           0x000000010a723189 $s9Endeavour17GoalTrackerScreenC11viewDidLoadyyF + 585
    18  Endeavour                           0x000000010a72321b $s9Endeavour17GoalTrackerScreenC11viewDidLoadyyFTo + 43
    19  UIKitCore                           0x00000001108c7f01 -[UIViewController
        _sendViewDidLoadWithAppearanceProxyObjectTaggingEnabled] + 83
    20  UIKitCore                           0x00000001108cce5a -[UIViewController loadViewIfRequired] + 1084
    21  UIKitCore                           0x00000001108311e4 -[UINavigationController
        _updateScrollViewFromViewController:toViewController:] + 160
    22  UIKitCore                           0x00000001108314e8 -[UINavigationController
        _startTransition:fromViewController:toViewController:] + 144
    23  UIKitCore                           0x00000001108323b6 -[UINavigationController
        _startDeferredTransitionIfNeeded:] + 868
    24  UIKitCore                           0x0000000110833721 -[UINavigationController __viewWillLayoutSubviews] + 150
    25  UIKitCore                           0x0000000110814553 -[UILayoutContainerView layoutSubviews] + 217
    26  UIKitCore                           0x00000001114314bd -[UIView(CALayerDelegate) layoutSublayersOfLayer:] + 2478
    27  QuartzCore                          0x0000000112d47db1 -[CALayer layoutSublayers] + 255
    28  QuartzCore                          0x0000000112d4dfa3 _ZN2CA5Layer16layout_if_neededEPNS_11TransactionE + 517
    29  QuartzCore                          0x0000000112d598da
        _ZN2CA5Layer28layout_and_display_if_neededEPNS_11TransactionE + 80
    30  QuartzCore                          0x0000000112ca0848 _ZN2CA7Context18commit_transactionEPNS_11TransactionEd +
        324
    31  QuartzCore                          0x0000000112cd5b51 _ZN2CA11Transaction6commitEv + 643
    32  UIKitCore                           0x0000000110f63575 __34-[UIApplication _firstCommitBlock]_block_invoke_2 + 81
    33  CoreFoundation                      0x000000010ca8029c __CFRUNLOOP_IS_CALLING_OUT_TO_A_BLOCK__ + 12
    34  CoreFoundation                      0x000000010ca7fa08 __CFRunLoopDoBlocks + 312
    35  CoreFoundation                      0x000000010ca7a894 __CFRunLoopRun + 1284
    36  CoreFoundation                      0x000000010ca7a066 CFRunLoopRunSpecific + 438
    37  GraphicsServices                    0x00000001169e5bb0 GSEventRunModal + 65
    38  UIKitCore                           0x0000000110f4bd4d UIApplicationMain + 1621
    39  Endeavour                           0x000000010a7219ab main + 75
    40  libdyld.dylib                       0x000000010f308c25 start + 1
)
libc++abi.dylib: terminating with uncaught exception of type NSException
```

The issue seems to be within the register button's constraints. After checking my code, I found the cause of the crash. I forgot to add loginRegisterButton() as a subview to the screen but tried to set up its constraints regardless. I've now called the necessary function:

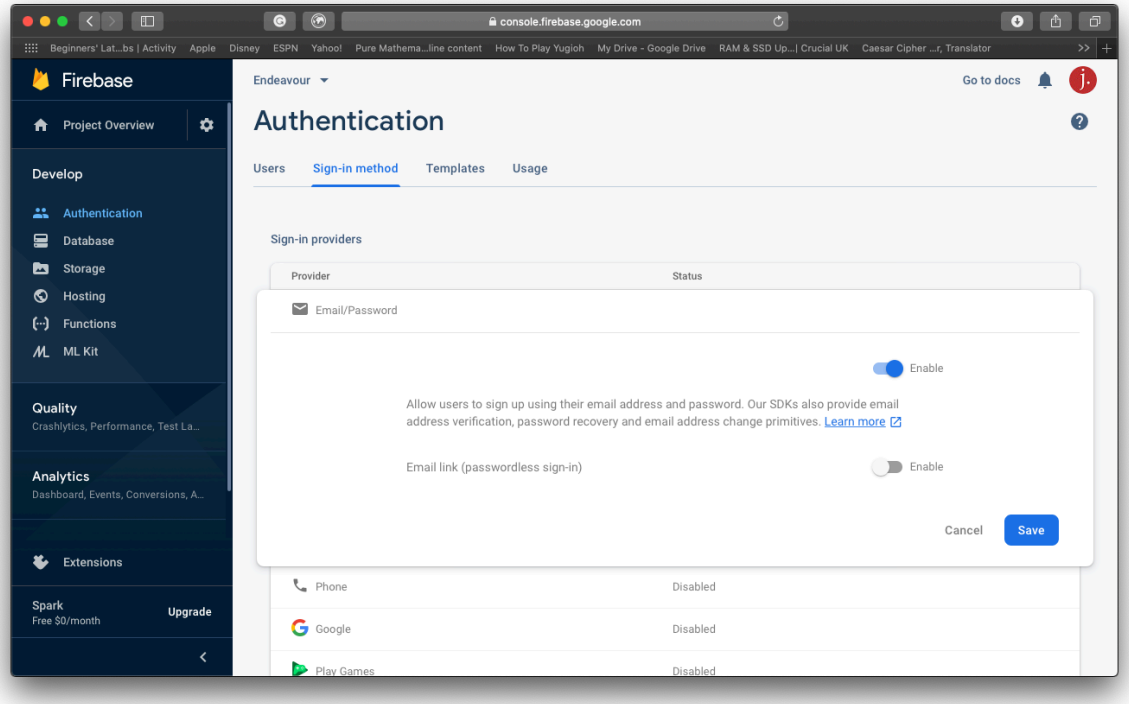
```
view.addSubview(loginRegisterButton)
```

The code now builds with no errors. This is what the login page looks like now:



Connecting the login page to Firebase

The first thing I will do is enable email and password authentication for the application's firebase database:

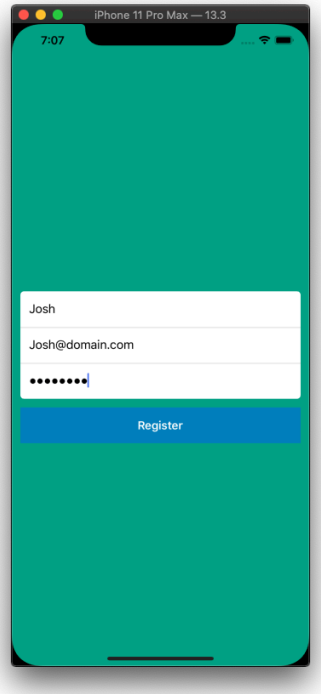


Now I can work on adding some code to the handleRegisterButton():

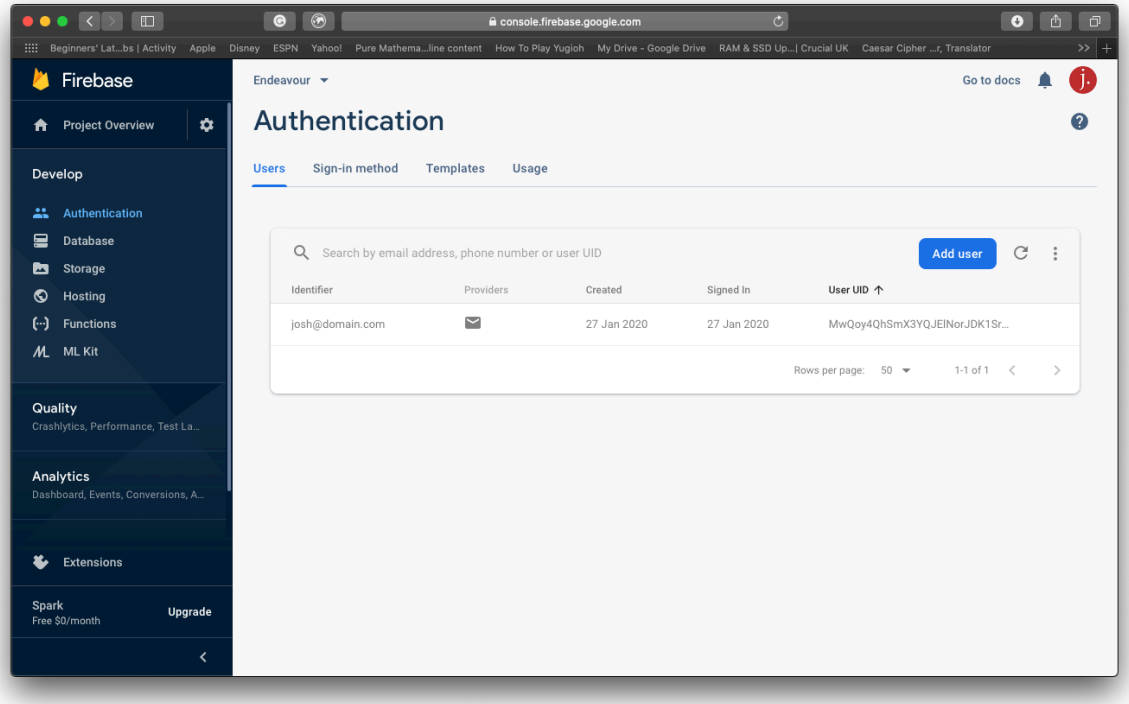
```
105     @objc func handleRegister() {
106         //This checks the inputs for email and password are actually valid
107         guard let email = emailTextField.text, let password = passwordTextField.text else {
108             print("Form is not valid")
109             return
110         }
111
112         //This creates a Firebase user using the inputs in the email and password fields.
113         Auth.auth().createUser(withEmail: email, password: password) { authResult, error in
114             //This if statement is there to catch any errors
115             if let err = error {
116                 print(err)
117             } else {
118
119             }
120         }
121
122     }
```

First of all, I need to collect the input from the various text fields to authenticate the user. Only the email and password are needed for the purposes of authentication so I'll exclusively work with those for the time being. The guard keyword allows me to assign the new variables values from the text fields at the same time as checking for basic validity.

The next part of the function is the code that actually interacts with Firebase. It uses the email and password taken from the input fields and then creates a new authenticated user in the firebase database. It also checks for any errors and stops the function if it catches them.



After adding this function, I tried inputting data into the text fields again before pressing register. Sure enough, a user was authenticated on Firebase:



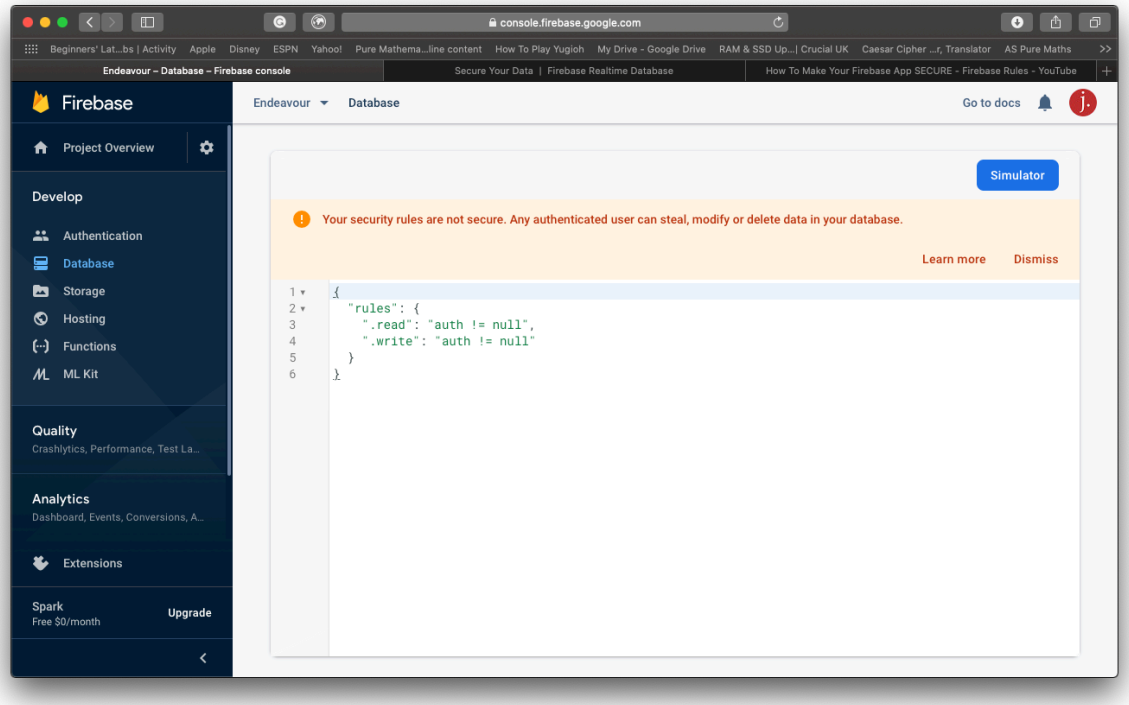
Now that I've sorted out user authentication, I can focus on writing the user's data to the database.

```
@objc func handleRegister() {
    //This checks the inputs for email and password are actually valid
    guard let email = emailTextField.text, let password = passwordTextField.text, let name = nameTextField.text else {
        print("Form is not valid")
        return
    }

    //This creates a Firebase user using the inputs in the email and password fields.
    Auth.auth().createUser(withEmail: email, password: password) { authResult, error in
        //This if statement is there to catch any errors
        if let err = error {
            print(err)
            return
        }

        //Save the new user to the database
        //Here, we create a variable that will reference the database.
        var ref: DatabaseReference!
        ref = Database.database().reference(fromURL: "https://endeavour-c6824.firebaseio.com/")

        //Create a dictionary of the values we want to store in the database
        let values: Dictionary = ["name": name, "email": email]
        let userID = Auth.auth().currentUser!.uid

        //Add the new user's name and email to the database
        //to the database beneath the child nodes "Users"
        //and a unique id generated by childByAutoId
        ref.child("Users").child(userID).updateChildValues(values, withCompletionBlock: {
            (error, ref) in

            if error != nil {
                print(error!)
                return
            }

            print("Saved user sccessfully")
        })

    }
}
```

First of all, I've created a new variable to hold the database reference called "ref". I've then made a dictionary called "values" to store the user's username and email address and I've assigned the constant "userID" the value of there auto-generated Firebase user ID. I then write the "values" dictionary to a subbranch of the node "Users" which is named after the corresponding user's unique id. There is also a little if statement that catches and prints out any errors.

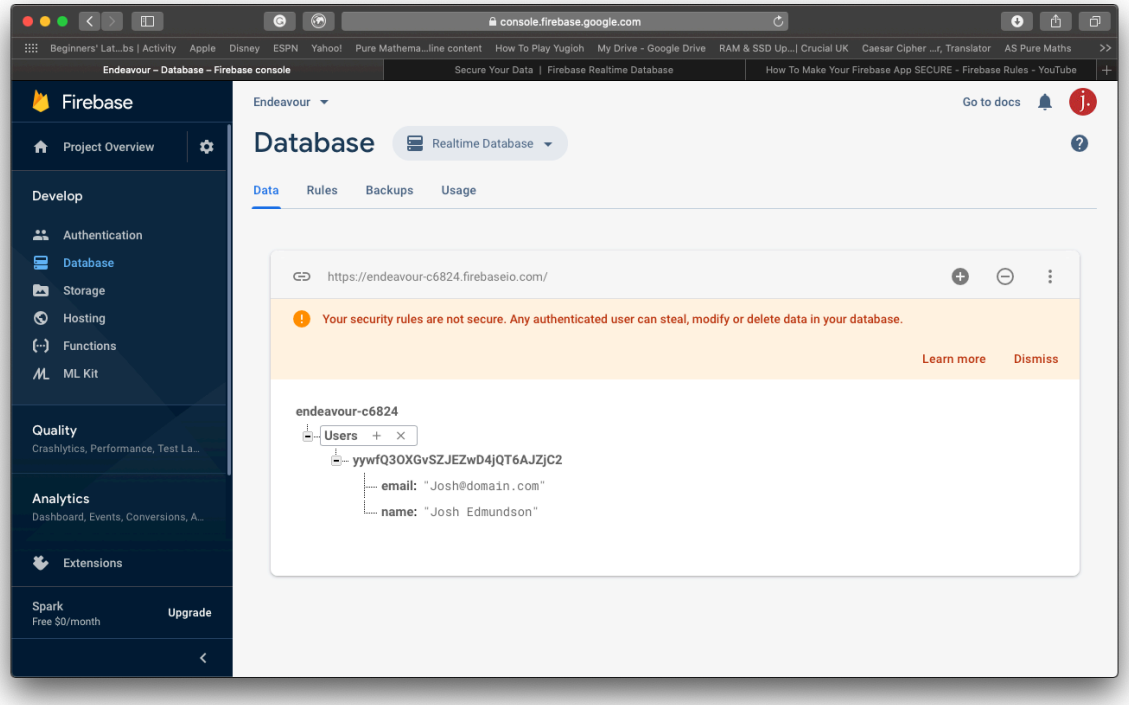
After intitially running this code, I kept getting a "permission denied" error. It seemed that even though I had authenticated several users, none of them could actually write information to the database. After much online searching, I finally found a way to solve the

problem. I edited the rules for the database on the Firebase website to allow any authenticated user to read and write information to the database.



The only issues is these rule updates are not very secure as it allows all users to read and write any data on the database so I will need to modify these rules at a later date.

The database structure now looks like this:



Allowing both logging in and registering

So far, users can only register to use the app. But what if they already have an account? What I want them to be able to do is be able to select whether they want to log in or create a new account.

The first thing I will do is add a segmented control:

```
87      //Create segmented controller
88      let loginRegisterSegmentedControl: UISegmentedControl = {
89          let sc = UISegmentedControl(items: ["Login", "Register"])
90          sc.translatesAutoresizingMaskIntoConstraints = false
91          sc.tintColor = UIColor.white
92          sc.selectedSegmentIndex = 1
93          return sc
94      }()
```

The above code creates a new UISegmentedControl object and sets up some of its basic attributes such as the names of its tabs.

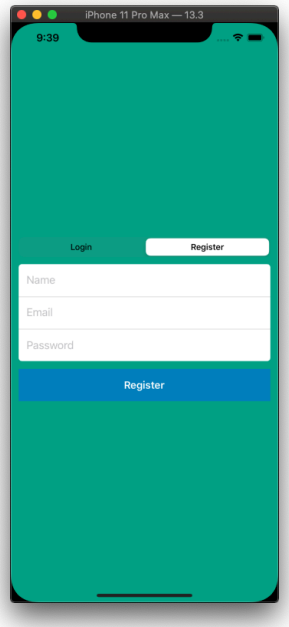
Next, I've created the constraints for the segmented control object:

```
159     //Set up constraints for loginRegisterSegmentedControl
160     func setupLoginRegisterSegmentedControl() {
161         //Setup constraints
162         loginRegisterSegmentedControl.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
163         loginRegisterSegmentedControl.bottomAnchor.constraint(equalTo: inputsContainerView.topAnchor, constant: -12).isActive = true
164         loginRegisterSegmentedControl.widthAnchor.constraint(equalTo: inputsContainerView.widthAnchor).isActive = true
165         loginRegisterSegmentedControl.heightAnchor.constraint(equalToConstant: 30).isActive = true
166     }
```

And finally, I've added it as a subview to the login/register page and called the setupLoginRegisterSegmentControl() function:

```
97      override func viewDidLoad() {
98          super.viewDidLoad()
99
100         //Change the background colour to "Green Sea"
101         view.backgroundColor = UIColor(displayP3Red: 22/255, green: 160/255, blue: 133/255, alpha: 1.0)
102
103         //Add all subviews to the screen
104         view.addSubview(inputsContainerView)
105         view.addSubview(loginRegisterButton)
106         view.addSubview(loginRegisterSegmentedControl)
107
108
109         //Create subview dimensions using constraints
110         setupInputsContainerView()
111         setupLoginRegisterButton()
112         setupLoginRegisterSegmentedControl()
113     }
```

Now when I build and run the application, this is what I get:



Note, currently toggling the segmented control doesn't actually change anything about the layout. Let's give the toggle a target to resolve that issue.

```
87      //Create segmented controller
88      let loginRegisterSegmentedControl: UISegmentedControl = {
89          let sc = UISegmentedControl(items: ["Login", "Register"])
90          sc.translatesAutoresizingMaskIntoConstraints = false
91          sc.tintColor = UIColor.white
92          sc.selectedSegmentIndex = 1
93          sc.addTarget(self, action: #selector(handleLoginRegisterChange), for:
                .valueChanged)
94          return sc
95      }()
```

Here, I've added line 93. This tells the segmented control to call the function handleLoginRegisterChange() when it is toggled. I then created the handleLoginRegisterChange() function:

```
117     //This is called when switiching between segments in the segmented control.
118     @objc func handleLoginRegisterChange() {
119         //Set up two variables base on the current segment selected
120         let currentIndex = loginRegisterSegmentedControl.selectedSegmentIndex
121         let title = loginRegisterSegmentedControl.titleForSegment(at: currentIndex)
122
123         //Update the text on the loginRegisterButton accordingly
124         loginRegisterButton.setTitle(title, for: .normal)
125
126         //Change height of inputContainerView based on the current index
127         inputsContainerViewHeightAnchor?.constant = currentIndex == 0 ? 100 : 150
128
129         //Change height of nameTextField
130         nameTextFieldHeightAnchor?.isActive = false
131         nameTextFieldHeightAnchor = nameTextField.heightAnchor.constraint(equalTo:
                inputsContainerView.heightAnchor, multiplier: currentIndex == 0 ? 0 : 1/3)
132         nameTextFieldHeightAnchor?.isActive = true
133
134         //Change the height of the emailTextField
135         emailTextFieldHeightAnchor?.isActive = false
136         emailTextFieldHeightAnchor = emailTextField.heightAnchor.constraint(equalTo:
                inputsContainerView.heightAnchor, multiplier: currentIndex == 0 ? 1/2 :
                1/3)
137         emailTextFieldHeightAnchor?.isActive = true
138
139         //Change the height of the passwordTextField
140         passwordTextFieldHeightAnchor?.isActive = false
141         passwordTextFieldHeightAnchor =
                passwordTextField.heightAnchor.constraint(equalTo:
                inputsContainerView.heightAnchor, multiplier: currentIndex == 0 ? 1/2 :
                1/3)
142         passwordTextFieldHeightAnchor?.isActive = true
143     }
```

There is a lot going on in this method. The first thing I did was create a couple of constants to hold information about the state of the segmented control such as its current index (0 or 1 depending on the selected segment) and the title of its current index (Either "Login" or "Register").

Line 124 changes the text in the loginRegisterButton when the segment is toggled.

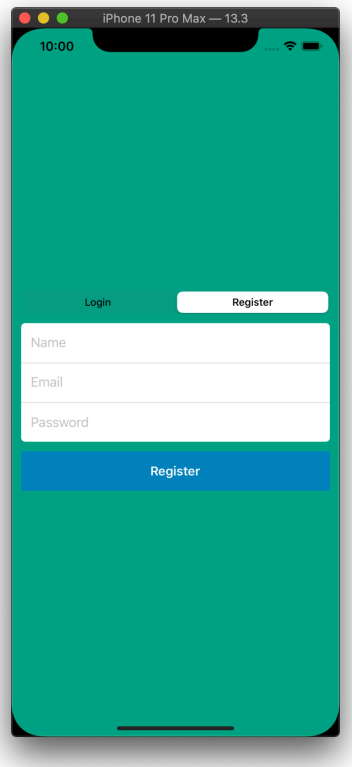
When the user selects the option to login, they don't need to present their username. The only information Firebase needs to confirm they are indeed a user is their email and password. Because of this, when they toggle the segmented controller to the "Login" position, I have to remove the nameTextField object from view. To do this, I created several new variables and inserted them into the constraints of the text field objects.

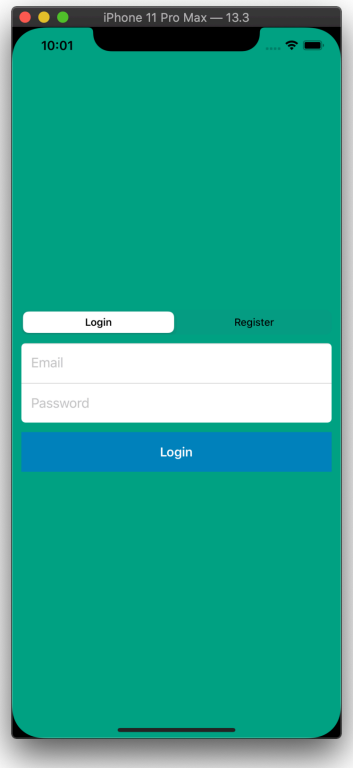
```
199     //Create variables that can change the layout constraints of the view
200     //and that can be accessed outisde of setupInputsContainerView()
201     var inputsContainerViewHeightAnchor: NSLayoutConstraint?
202     var nameTextFieldHeightAnchor: NSLayoutConstraint?
203     var emailTextFieldHeightAnchor: NSLayoutConstraint?
204     var passwordTextFieldHeightAnchor: NSLayoutConstraint?
```

```
228         //Set up the constraints for nameTextField
229         nameTextField.leftAnchor.constraint(equalTo: inputsContainerView.leftAnchor,
                constant: 12).isActive = true
230         nameTextField.topAnchor.constraint(equalTo:
                inputsContainerView.topAnchor).isActive = true
231         nameTextField.widthAnchor.constraint(equalTo:
                inputsContainerView.widthAnchor).isActive = true
232         //Place the height anchor wihin the variable nameTextFieldHeightAnchor
233         nameTextFieldHeightAnchor = nameTextField.heightAnchor.constraint(equalTo:
                inputsContainerView.heightAnchor, multiplier: 1/3)
234         nameTextFieldHeightAnchor?.isActive = true
```

I can then use these variables within the handleLoginRegisterChange() method to alter the dimensions of the textFields on the screen (along with the dimensions of inputsContainerView).

The result is two separate displays:





I can then add one line of code to dismiss the login/register view when the user successfully makes an account:

```
182                 self.dismiss(animated: true, completion: nil)
```

Currently, the user can only create a new account, they cannot actually sign in with pre-existing information.

```
146     //This function determins whether the user is trying to login
147     //or register and runs the corresponding function
148     @objc func handleLoginRegister() {
149         if loginRegisterSegmentedControl.selectedSegmentIndex == 0 {
150             handleLogin()
151         } else {
152             handleRegister()
153         }
154     }
155
156
157     //This function will sign the user in.
158     func handleLogin() {
159         guard let email = emailTextField.text, let password = passwordTextField.text
                else {
160             print("Form is not valid")
161             return
162         }
163
164         Auth.auth().signIn(withEmail: email, password: password, completion: {
165             (user, error) in
166
167             if error != nil {
168                 print(error as Any)
169                 return
170             }
171
172             self.dismiss(animated: true, completion: nil)
173
174         })
175     }
```

I've gone and written two new methods that will now allow the user to either login or sign in depending on the segmented control's index. The first function, on line 148, is run when the user presses the loginRegisterButton. If the index has a value of 0, the "if" statement will call handleLogin(). If the index is equal to 1, then handleRegister() is called. I've added the self.dismiss(animated: true, completion: nil) line to both the handleRegister() and the handleLogin() functions so the screen self dismisses when the user logs in / creates an account.

In both the navigator bar of the goal tracking screen and the chat screen I've add a "Logout" button:

```
15      override func viewDidLoad() {
16          super.viewDidLoad()
17
18          //Make the background colour white
19          view.backgroundColor = .white
20
21          title = "Goals"
22
23          //Add a logout button to the left of the navigation bar
24          navigationItem.leftBarButtonItem = UIBarButtonItem(title: "Logout", style:
                .plain, target: self, action: #selector(displayLoginRegisterScreen))
25
26          //Checks to see if the user is logged in already
27          if Auth.auth().currentUser?.uid == nil {
28              //If the user is not logged in, display the login page
29              displayLoginRegisterScreen()
30          }
31
32      }
```

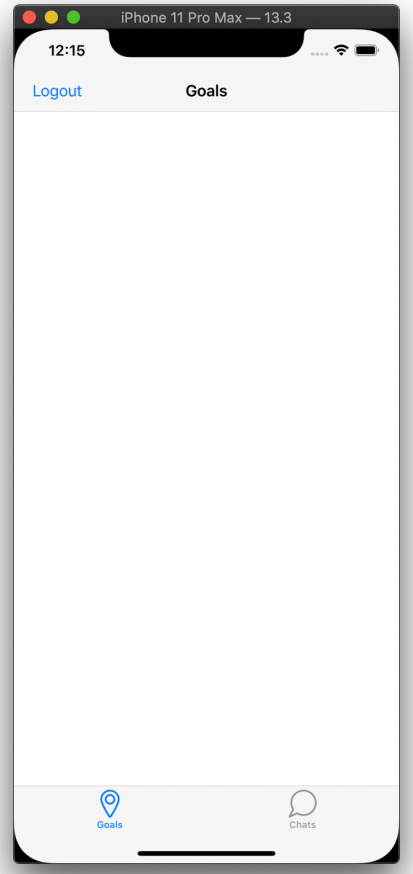
In both the GoalTrackerScreen and ChatScreen viewDidLoad() functions, I've also added an "if" statement that checks if the current user has an assigned unique ID (see line 27). When a user creates an account in this app, they are automatically given an ID by Firebase. If the current user is not logged into an authenticated account, they will have an ID value of 'nil'. The "if" statement therefore registers if the user is logged in or not. If they aren't, then the login page is displayed. I've used this snippet of code to replace the previous method of displaying the loginRegisterScreen which would push the screen to the top of the view stack regardless of whether the user had remained logged in or not.

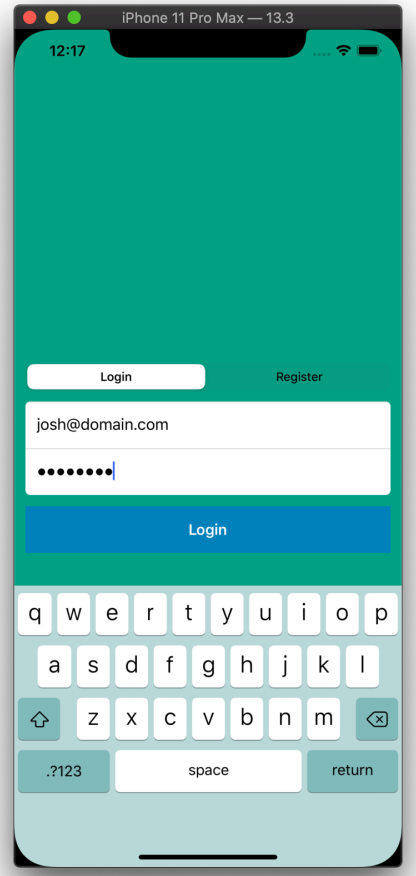
As a result of using a logout button, I wrote a function to handle the logout called logOutCurrentUser() which is called by the displayLoginRegisterScreen().

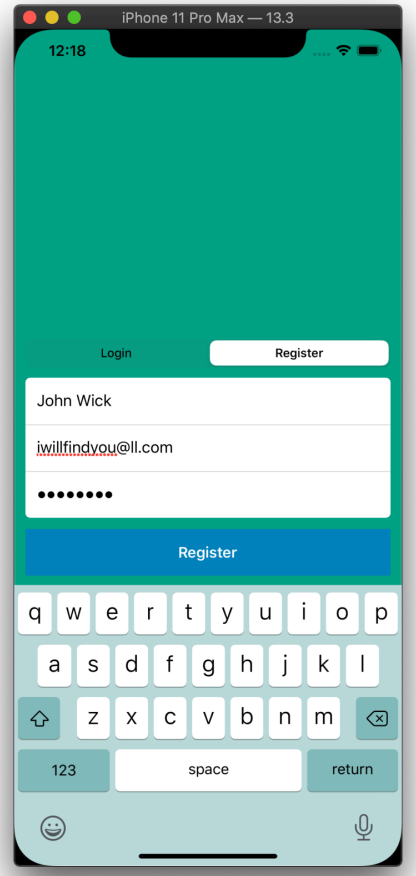
```
35      //This function can be used to display the login in/register screen
36      @objc func displayLoginRegisterScreen() {
37          //Log the user out of firebase
38          logOutCurrentUser()
39
40          //Instansiate LoginRegisterScreen
41          let loginRegisterScreen = LoginRegisterScreen()
42
43          //Set the display to fullscreen
44          loginRegisterScreen.modalPresentationStyle = .fullScreen
45
46          //Display the LoginRegisterScreen object
47          present(loginRegisterScreen, animated: true, completion: nil)
48      }
49
50
51      //This logs the current user out of Firebase
52      func logOutCurrentUser() {
53          do {
54              try Auth.auth().signOut()
55          } catch let logoutError {
56              print(logoutError)
57          }
58      }
```

The logout function tries to log out the user but will also catch itself if the user is not actually logged in in the first place. This allows the user to both logout of Firebase and to redisplay the login screen at the click of a button.

This is now what the app looks like:







Setting up contacts

Creating a “New Message” screen

If a user is going to be able to send messages, they will need to be able to actually select other users they wish to talk to. Currently in the app, there is no way to do that. Let’s start by pulling user information from Firebase into the app:

```
55      //Gets the current user's information when called
56      func getCurrentUsersInfo() {
57          let uid = Auth.auth().currentUser?.uid
58          Database.database().reference().child("Users").child(uid!).observeSingleEvent(of: .value, with: { (snapshot) in
59
60              print(snapshot)
61
62          }, withCancel: nil)
63
64      }
```

I’ve written this function within “ChatScreen.swift”. Is listens to the value(s) found at the root of the specified nodes. Here, I’m telling the app to retrieve any information about the user found at the node that shares their unique user ID. This will return their name and email.

To check the above function actually works, I've called it in the viewDidLoad() method and this is the output:

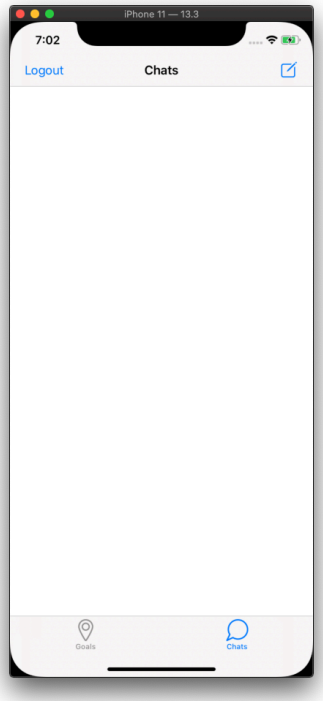
```
2020-02-04 06:58:19.903388+0000 Endeavour[2163:122592] 6.15.0 - [Firebase/Analytics][I-ACS023012]
    Analytics collection enabled
Snap (yywfQ3OXGvSZJEZwD4jQT6AJZjC2) {
    email = "Josh@domain.com";
    name = "Josh Edmundson";
}
```

Everything is running smoothly.

I've added a new button to the navigation controller in the chat screen to allow the user to compose a new message:

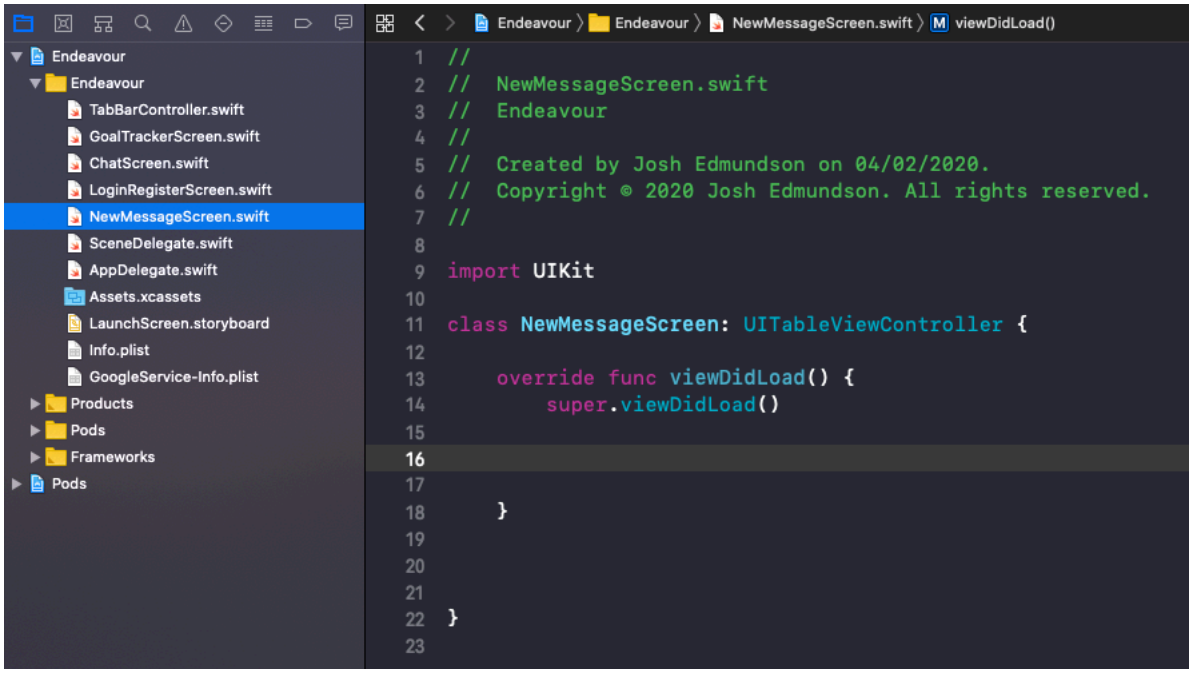
```
26          navigationItem.rightBarButtonItem = UIBarButtonItem(barButtonSystemItem: .compose, target: self, action: .none)
```

Now when I run the simulator, I this is what the chat screen looks like:



Right now this button doesn't do anything when you click on it. Let's give it a function to execute.

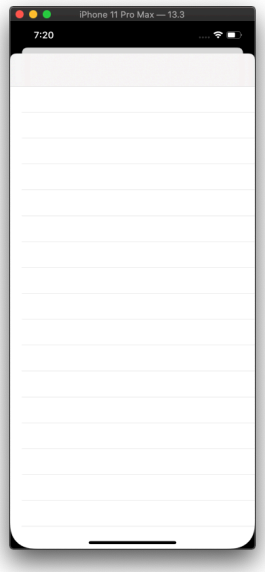
First thing's first, I've created a new class called "NewMessageScreen.swift". When the user clicks the "compose" button, the app will display an instance of this class. NewMessageScreen inherits from UITableViewController, meaning I will be able to manipulate any instance of NewMessageScreen like a table, allowing me to insert cells of information. This is how the user's contacts will be displayed.



Within the ChatScreen class, I've also created a function that I've linked to the navigation button called handleNewMessage.

```
46      //This function will display an instance of the UINavigationController class within a
47      //root view controller that is an instance of NewMessageScreen.
48      @objc func handleNewMessage() {
49          let newMessageScreen = NewMessageScreen()
50          let navigationController = UINavigationController(rootViewController: newMessageScreen)
51          present(navigationController, animated: true, completion: nil)
52      }
```

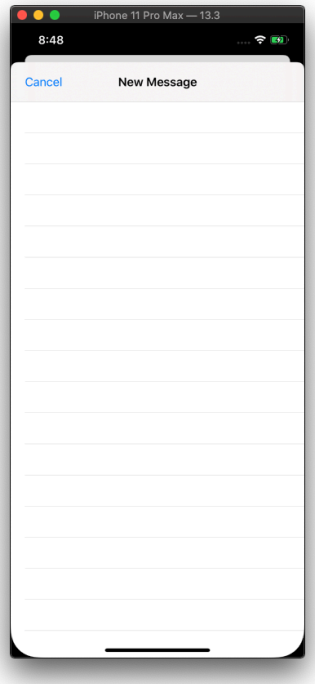
This creates an instance of the NewMessageScreen class, gives it a navigation bar and then presents it to the user. I've deliberately not specified the navigationController as being full screen. That way the user can dismiss the compose message window by simply swiping down on the screen.



I've also added a "Cancel" button to the composeMessage navigation bar:

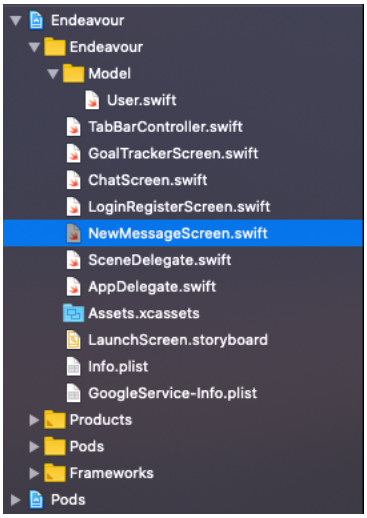
```swift
import UIKit

class NewMessageScreen: UITableViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        //Adds the cancel button to the navigation bar
        navigationItem.leftBarButtonItem = UIBarButtonItem(barButtonSystemItem: .cancel, target: self, action: #selector(handleCancel))

        //Changes the title of the view controller
        title = "New Message"


    }


    //Dismisses the screen when the user presses the cancel button.
    @objc func handleCancel() {
        self.dismiss(animated: true, completion: nil)
    }



}
```

It calls the function handleCancel() when called, dismissing itself. I've also given the instance of NewMessageScreen a title.



Retrieving users from Firebase

I've created a new group called "Model" where I will store all custom classes and objects and within this new group, I have created a new class called "User".



This new class inherits from NSObject and has two attributes, name and email.

```swift
import UIKit

class User: NSObject {

    var name: String?
    var email: String?

}
```

Within NewMessageScreen, I have gone and made a list in which I will store the various users contained within the Firebase database.

```swift
var users = [User]()
```

This list will only store instances of the User class. Now I need to actually retrieve each user's information.

```swift
//When run, fetches users from firebase
func fetchUsers() {

    //Create a Firebase reference
    Database.database().reference().child("Users").observe(.childAdded, with: { (snapshot) in

        //Creates a variable "dictionary" and stores the values contained within the snapshot as a
        //key-value pair within the dictionary data structure.
        if let dictionary = snapshot.value as? [String: AnyObject] {
            //Creates an instance of User
            let user = User()

            //Gives the attributes of user values
            user.name = dictionary["name"] as? String
            user.email = dictionary["email"] as? String

            //Adds the new user to the list.
            self.users.append(user)

            print(user.name as Any, user.email as Any)
        }


    }, withCancel: nil)

}
```

This function creates a reference to the database and retrieves the information contained at each sub-node of “Users” as a snapshot. Each snapshot (each user’s unique ID) contains the user’s email and username. The function finds those values and stores them in a dictionary. Then, we instantiated the class “User” and modify its attributes to match the name and email retrieved from the snapshot. By the time the function has finished running, we have added every user from the online database into the “users” list.

```
Optional("Test One") Optional("Test@one.com")
Optional("John Wick") Optional("iwillfindyou@ll.com")
Optional("Josh Edmundson") Optional("Josh@domain.com")
```

The print statement at the end of the function outputs the above to the console. As we can see, the code has successfully retrieved the relevant user information from the online database.

Displaying users in the table view

Now that we have successfully retrieved each user’s name and email from Firebase, we have to display them within the table view.

I’ve overridden a function inherited from the UITableView class to set the number of rows within the table view to equal the number of users in the “users” list:

```
70      //Overides the tableView function from the UITableViewController parent class and sets
71      //the number of rows to be displayed with information to the number of users.
72      override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
73          return users.count
74      }
```

I’ve then created a new class called “UserCell” as a sub-class of UITableViewCell that will allow for more flexability when it comes to customising how I want the user information to be displayed.

```
93  //Creates a custom cell class that allows the program to display the usernames
94  //with the format .subtitle
95  class UserCell: UITableViewCell {
96
97      override init(style: UITableViewCell.CellStyle, reuseIdentifier: String?) {
98          super.init(style: .subtitle, reuseIdentifier: reuseIdentifier)
99      }
100
101     required init?(coder: NSCoder) {
102         fatalError("init(coder:) has not been implemented")
103     }
104 }
```

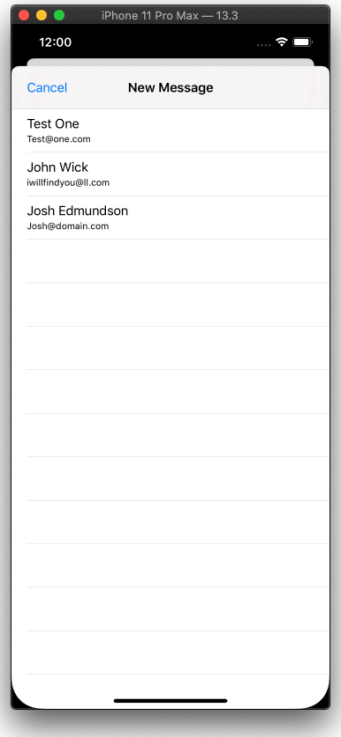
I’ve then had to register this custom cell class within the viewDidLoad() function:

```
28          tableView.register(UserCell.self, forCellReuseIdentifier: cellID)
```

The final function I’ve written creates a cell, using the aforementioned custom class, for each row in the table. Each cell contains the name and email address of a user from the databse.

```swift
77      //A custom cell for each row we are displaying and displays each user's name and email on the relevant cell.
78      override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
79
80          let cell = tableView.dequeueReusableCell(withIdentifier: cellID, for: indexPath)
81
82          let user = users[indexPath.row]
83
84          cell.textLabel?.text = user.name
85          cell.detailTextLabel?.text = user.email
86
87          return cell
88      }
```

And there we have it. When I build and run the application:



Here is all the code now in the NewMessageScreen class:

```swift
import UIKit
import Firebase

class NewMessageScreen: UITableViewController {

    let cellID = "cellID"

    var users = [User]()

    override func viewDidLoad() {
        super.viewDidLoad()

        //Adds the cancel button to the navigation bar
        navigationItem.leftBarButtonItem = UIBarButtonItem(barButtonSystemItem: .cancel, target: self, action: #selector(handleCancel))

        //Changes the title of the view controller
        title = "New Message"

        //Registers our custom cell class.
        tableView.register(UserCell.self, forCellReuseIdentifier: cellID)

        //Fetch the users from Firebase
        fetchUsers()

    }


    //When run, fetches users from firebase
    func fetchUsers() {

        //Create a Firebase reference
        Database.database().reference().child("Users").observe(.childAdded, with: { (snapshot) in

            //Creates a variable "dictionary" and stores the values contained within the snapshot as a
            //key-value pair within the dictionary data structure.
            if let dictionary = snapshot.value as? [String: AnyObject] {
                //Creates an instance of User
                let user = User()

                //Gives the attributes of user values
                user.name = dictionary["name"] as? String
                user.email = dictionary["email"] as? String

                //Adds the new user to the list.
                self.users.append(user)

                self.tableView.reloadData()
            }


        }, withCancel: nil)
    }


    //Dismisses the screen when the user presses the cancel button.
    @objc func handleCancel() {
        self.dismiss(animated: true, completion: nil)
    }


    //Overides the tableView function from the UITableViewController parent class and sets
    //the number of rows to be displayed with information to the number of users.
    override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return users.count
    }


    //A custom cell for each row we are displaying and displays each user's name and email on the relevant cell.
    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

        let cell = tableView.dequeueReusableCell(withIdentifier: cellID, for: indexPath)

        let user = users[indexPath.row]

        cell.textLabel?.text = user.name
        cell.detailTextLabel?.text = user.email

        return cell
    }

}


//Creates a custom cell class that allows the program to display the usernames
//with the format .subtitle
class UserCell: UITableViewCell {

    override init(style: UITableViewCell.CellStyle, reuseIdentifier: String?) {
        super.init(style: .subtitle, reuseIdentifier: reuseIdentifier)
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

Sending messages

Creating the interface

I've created a new view controller called ChatLogController which I will build into the messaging interface.

```swift
import UIKit

class ChatLogController: UITableViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

    }
}

```

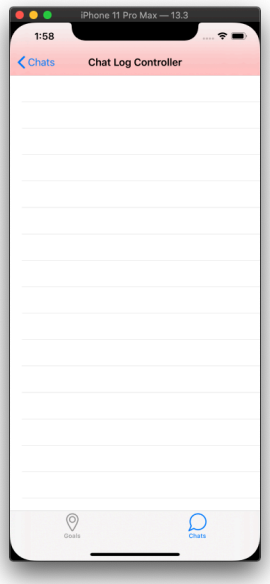
I've temporarily change the function of the "Compose" button on the ChatScreen view controller so that if pushes an instance of ChatLogController when pressed instead of an instance of NewMessageScreen. This way, I can easily test the ChatLogController as I go.

```
30      //Function that will display chatLogController when compose button is clicked. Remove later.
31      @objc func displayChatLogController() {
32          let chatLogController = ChatLogController()
33          navigationController?.pushViewController(chatLogController, animated: true)
34      }
```

Let's start adding the input field that will stay at the bottom of the screen:

```
9   import UIKit
10
11  class ChatLogController: UITableViewController {
12      override func viewDidLoad() {
13          super.viewDidLoad()
14
15          title = "Chat Log Controller"
16
17          setupInputComponents()
18      }
19
20      func setupInputComponents() {
21          let containerView = UIView()
22          containerView.backgroundColor = UIColor.red
23          containerView.translatesAutoresizingMaskIntoConstraints = false
24
25          view.addSubview(containerView)
26
27          //constraints
28          containerView.leftAnchor.constraint(equalTo: view.leftAnchor).isActive = true
29          containerView.widthAnchor.constraint(equalTo: view.widthAnchor).isActive = true
30          containerView.bottomAnchor.constraint(equalTo: view.bottomAnchor).isActive = true
31          containerView.heightAnchor.constraint(equalToConstant: 50).isActive = true
32      }
33  }
34
```

I've simply created a new UIView and anchored it to the bottom of the screen. However, when I run the application, this happens:



The UIView that I've just made gets placed right at the top of the screen for some odd reason. One work around to this is to make ChatLogController a sub-class of UICollectionViewController instead of UITableViewController.

```
11  class ChatLogController: UICollectionViewController
```

When I build and run the project now, I get a different error:

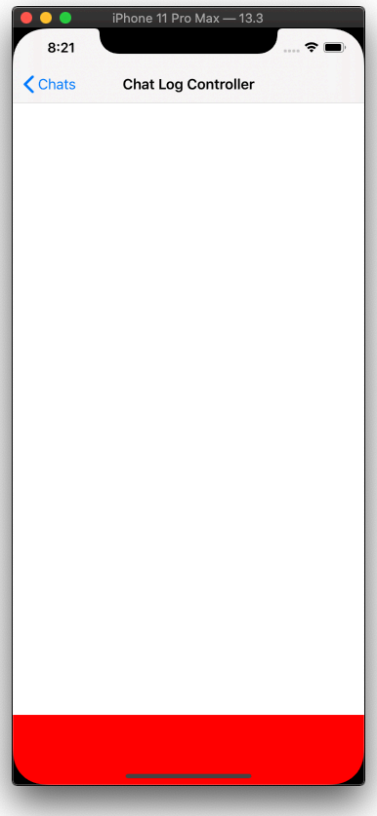
```
2020-02-09 14:00:39.965514+0000 Endeavour[3777:347785] *** Terminating app due to uncaught
    exception 'NSInvalidArgumentException', reason: 'UICollectionView must be initialized with a
    non-nil layout parameter'
```

To fix this, I modified the function in ChatScreen.swift that creates an instance of ChatViewController:

```
30      //Function that will display chatLogController when compose button is clicked. Remove later.
31      @objc func displayChatLogController() {
32          let chatLogController = ChatLogController(collectionViewLayout: UICollectionViewFlowLayout())
33          navigationController?.pushViewController(chatLogController, animated: true)
34      }
```

After then adding the code below to the displayChatLogController within ChatScreen.swift, we get the following:

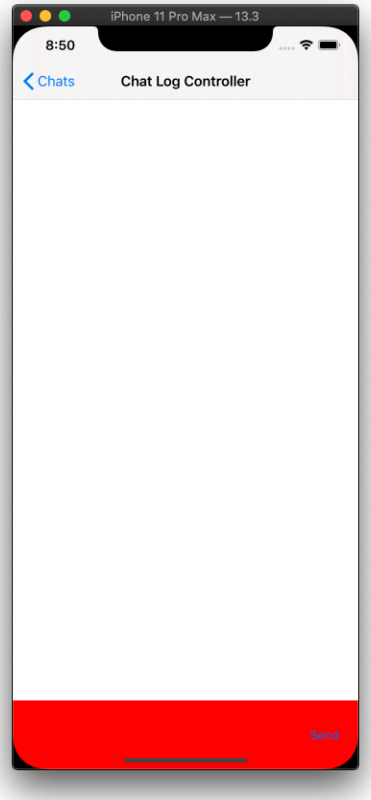
```
33          chatLogController.hidesBottomBarWhenPushed = true
```



The reason I added the above line of code was to hide the tab bar along the bottom of the screen when the instance of chatLogController is pushed to the view.

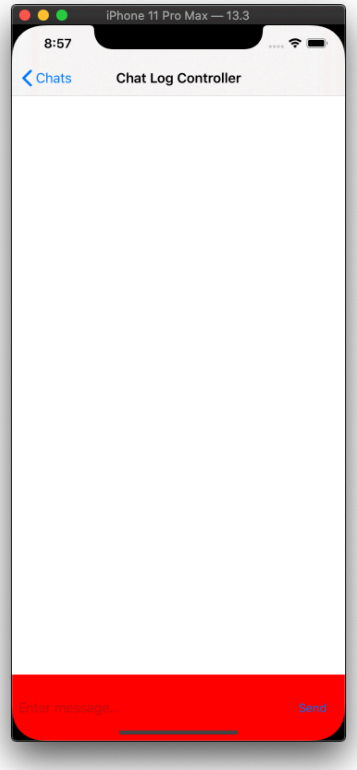
I've then added a send button to the containerView, still within the setupInputComponents function:

```
37          //Setup send button
38          let sendButton = UIButton(type: .system)
39          sendButton.setTitle("Send", for: .normal)
40          sendButton.translatesAutoresizingMaskIntoConstraints = false
41
42          containerView.addSubview(sendButton)
43
44          //Constrain sendButton
45          sendButton.rightAnchor.constraint(equalTo: containerView.rightAnchor).isActive = true
46          sendButton.centerYAnchor.constraint(equalTo: containerView.centerYAnchor).isActive = true
47          sendButton.widthAnchor.constraint(equalToConstant: 80).isActive = true
48          sendButton.heightAnchor.constraint(equalTo: containerView.heightAnchor).isActive = true
```



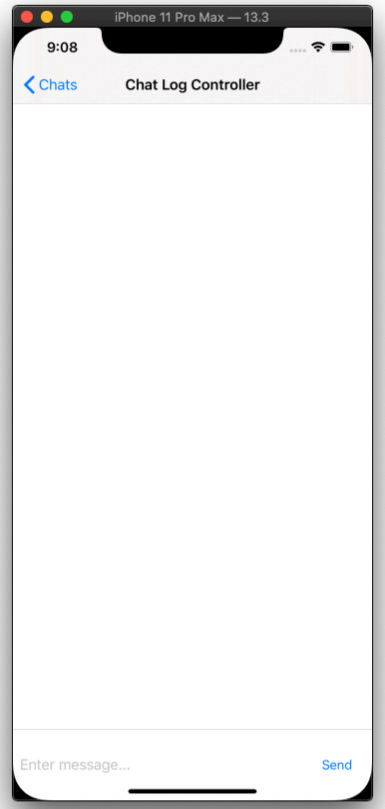
I've then added the a text input field to the containerView as a subview:

```
51          //Create a text input
52          let inputTextField = UITextField()
53          inputTextField.placeholder = "Enter message..."
54          inputTextField.translatesAutoresizingMaskIntoConstraints = false
55
56          containerView.addSubview(inputTextField)
57
58          //Constraints for inputTextField
59          inputTextField.leftAnchor.constraint(equalTo: containerView.leftAnchor, constant: 8).isActive = true
60          inputTextField.centerYAnchor.constraint(equalTo: containerView.centerYAnchor).isActive = true
61          inputTextField.rightAnchor.constraint(equalTo: sendButton.leftAnchor).isActive = true
62          inputTextField.heightAnchor.constraint(equalTo: containerView.heightAnchor).isActive = true
```



Finally, I've change the background colour of the viewContainer to white and I've added a line to separate the input text field and button from the rest of the view.

```
64          //Create separator to seperate the containerView from the rest of view
65          let separatorLineView = UIView()
66          separatorLineView.backgroundColor = UIColor(displayP3Red: 220/255, green: 220/255, blue: 220/255, alpha: 1)
67          separatorLineView.translatesAutoresizingMaskIntoConstraints = false
68
69          containerView.addSubview(separatorLineView)
70
71          //Constrain separatorLineView
72          separatorLineView.leftAnchor.constraint(equalTo: containerView.leftAnchor).isActive = true
73          separatorLineView.widthAnchor.constraint(equalTo: containerView.widthAnchor).isActive = true
74          separatorLineView.topAnchor.constraint(equalTo: containerView.topAnchor).isActive = true
75          separatorLineView.heightAnchor.constraint(equalToConstant: 1).isActive = true
```



Uploading the message to Firebase

Currently, if I enter text into the text field and hit send, nothing happens. To change this, I need to add a function that uploads the text entered into the text field to Firebase and then empty the text field.

First of all, I've had to change the way in which I created the inputTextField object:

```
14      //This reason we define inputTextField in this way is so that we can
15      //reference it outside of the setupInputComponents() function
16      let inputTextField: UITextField = {
17          let textField = UITextField()
18          textField.placeholder = "Enter message..."
19          textField.translatesAutoresizingMaskIntoConstraints = false
20          return textField
21      }()
```

I've had to define it outside of the scope of the setupInputComponents() method so that I can reference the text within it in other parts of the program.

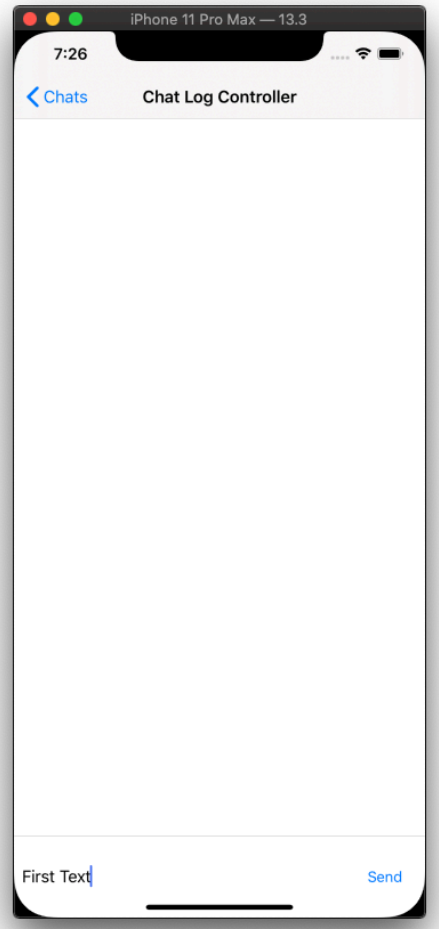
I've then gone and created a function that will be called whenever the "Send" button is pushed:

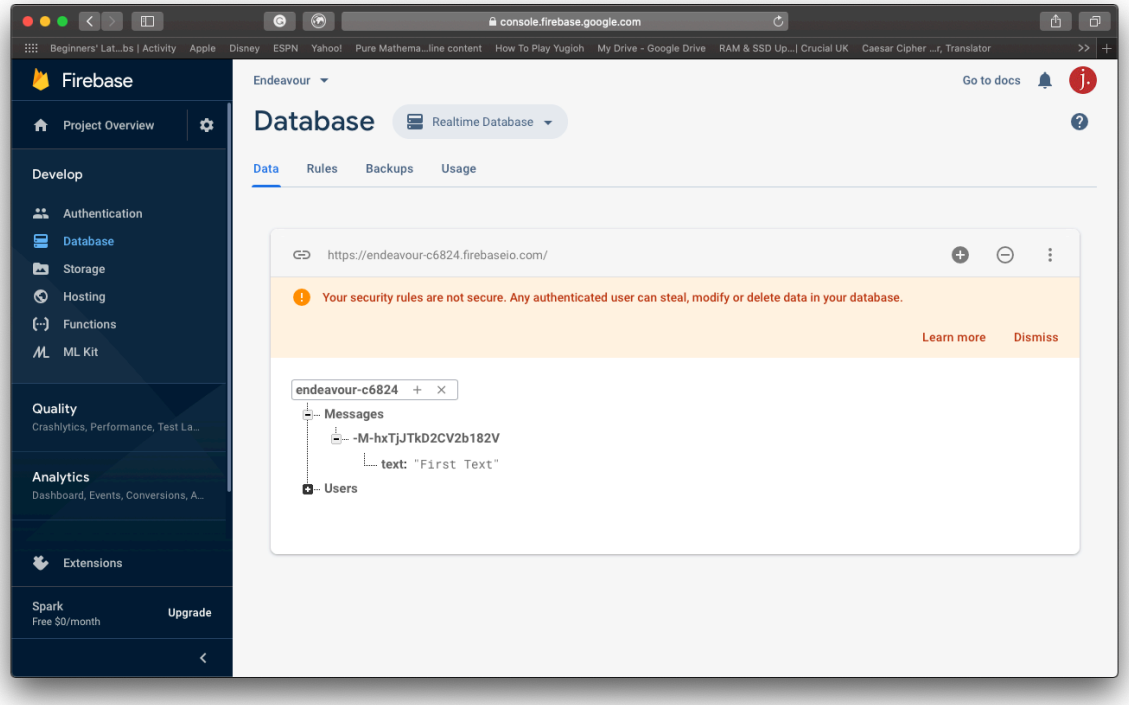
```
91      //Uploads the text from the inputTextField to the cloud when called
92      @objc func handleSend() {
93          //Creates a reference to a new node in firebase "Messages"
94          let ref = Database.database().reference().child("Messages")
95          //Creates a unique ID for this particular message node
96          let childRef = ref.childByAutoId()
97          //Uploads text from the text field to the database.
98          let values = ["text": inputTextField.text!]
99          childRef.updateChildValues(values)
100         //Clears the text field
101         inputTextField.text = ""
102     }
```

Finally, I've gone and added the line of code that links the above function to the "Send" button:

```
54          sendButton.addTarget(self, action: #selector(handleSend), for: .touchUpInside)
```

Now, when I build and run the project:





The first text has successfully been sent!

To allow the user to send a message by hitting the “Enter” key, I’ve modified the code a little bit.

```
12  class ChatLogController: UICollectionViewController, UITextFieldDelegate {
13
14      //This reason we define inputTextField in this way is so that we can
15      //reference it outside of the setupInputComponents() function
16      lazy var inputTextField: UITextField = {
17          let textField = UITextField()
18          textField.placeholder = "Enter message..."
19          textField.translatesAutoresizingMaskIntoConstraints = false
20          textField.delegate = self
21          return textField
22      }()
```

Above, I’ve added line 20, and changed the constant inputTextField to a lazy variable, giving inputTextField access to self. The class ChatLogController now also inherits from UITextFieldDelegate, allowing me to access the methods contained within it.

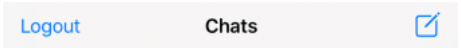
I’ve then added this function to the end of the file:

```
106     //Allows the user to send a message by pressing the "Enter" key
107     func textFieldShouldReturn(_ textField: UITextField) -> Bool {
108         handleSend()
109         return true
110     }
```

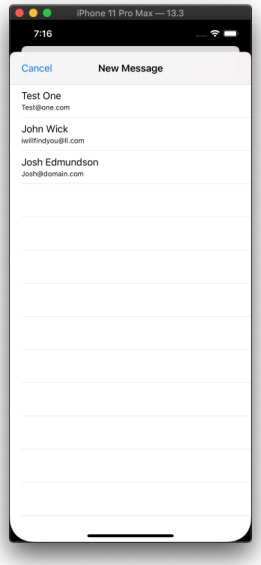
So now when I hit enter, it will send a message.

Sorting out control flow

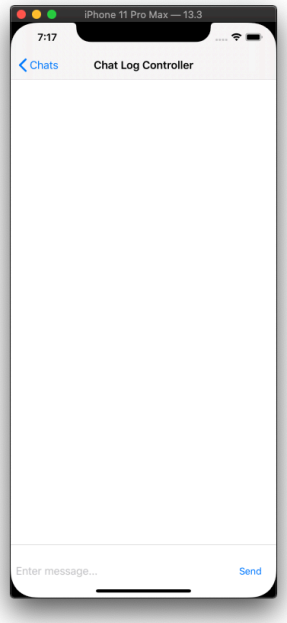
At the moment, the ChatLogController class is all set up and the app can successfully upload messages to Firebase. However, the only way to display an instance of ChatLogController at the moment is by clicking the compose button in the chat screen's navigation bar.



(The compose button is the one on the left.) When the user clicks the compose button, they need to be taken to and instance of NewMessageScreen instead of ChatLogController. This:



Instead of this:



The first thing I've done is change the function called when the compose button on the chat screen is clicked.

```
26          navigationItem.rightBarButtonItem = UIBarButtonItem(barButtonSystemItem: .compose, target: self, action: #selector(handleNewMessage))
```

This now displays the list of potential contacts when selected, as it did originally.

I've then created an instance of ChatScreen within the NewMessageScreen class so that I can access the displayChatLogController() function. I've also modified the inherited function tableView again to that when a cell is selected from the list of contacts, the app runs the displayChatLogController() function.

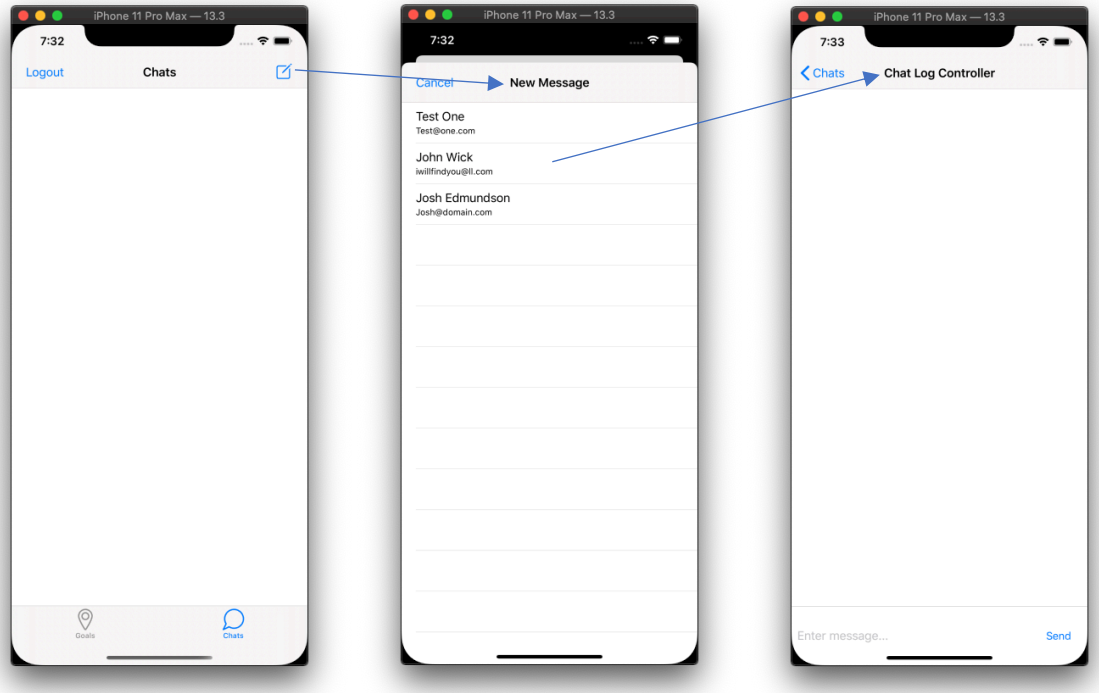
```
91      //Create an instance of chat screen so we can access the function displayChatLogController
92      var chatScreen: ChatScreen?
93
94      //Instruct the app to call displayChatLogController() when a cell is selected.
95      override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
96          dismiss(animated: true, completion: {
97              print("Completed")
98              self.chatScreen?.displayChatLogController()
99          })
100     }
```

Now, when I run this, and select a contact (say John Wick), the application doesn't create an instance of ChatLogController. To rectify this issue, I've had to modify the handleNewMessage() function within the ChatScreen class.

```
54      //This function will display an instance of the UINavigationController class within a
55      //root view controller that is an instance of NewMessageScreen.
56      @objc func handleNewMessage() {
57          let newMessageScreen = NewMessageScreen()
58          newMessageScreen.chatScreen = self
59          let navigationController = UINavigationController(rootViewController: newMessageScreen)
60          present(navigationController, animated: true, completion: nil)
61      }
```

I've added the highlighted line of code (line 58), giving the instance of NewMessageScreen's attribute chatScreen a value of self. This explicitly tells the application to run the function called displayChatLogController, contained within self, when a contact is selected.

Now, the control flow is sorted.



Displaying the contacts name

As you can see in the images above, when you select a user, it takes you to a view with the title “Chat Log Controller”. I want to change this so you can actually see the name of the user you are writing a message to.

In NewMessageScreen.swift, I’ve modified the tableView(…) function:

```
94      //Instruct the app to call displayChatLogController() when a cell is selected.
95      override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
96          dismiss(animated: true, completion: {
97              print("Completed")
98
99              //Sets the constant user equal to the user selected from the contacts list.
100             let user = self.users[indexPath.row]
101             //Calls displayChatLogControllerForUser() for the current selected user
102             self.chatScreen?.displayChatLogControllerForUser(user: user)
103         })
104     }
```

This sets the variable “user” equal to the user that has been selected. I’ve then modified the previous displayChatLogController function, now called displayChatLogControllerForUser, to take the parameter “user”.

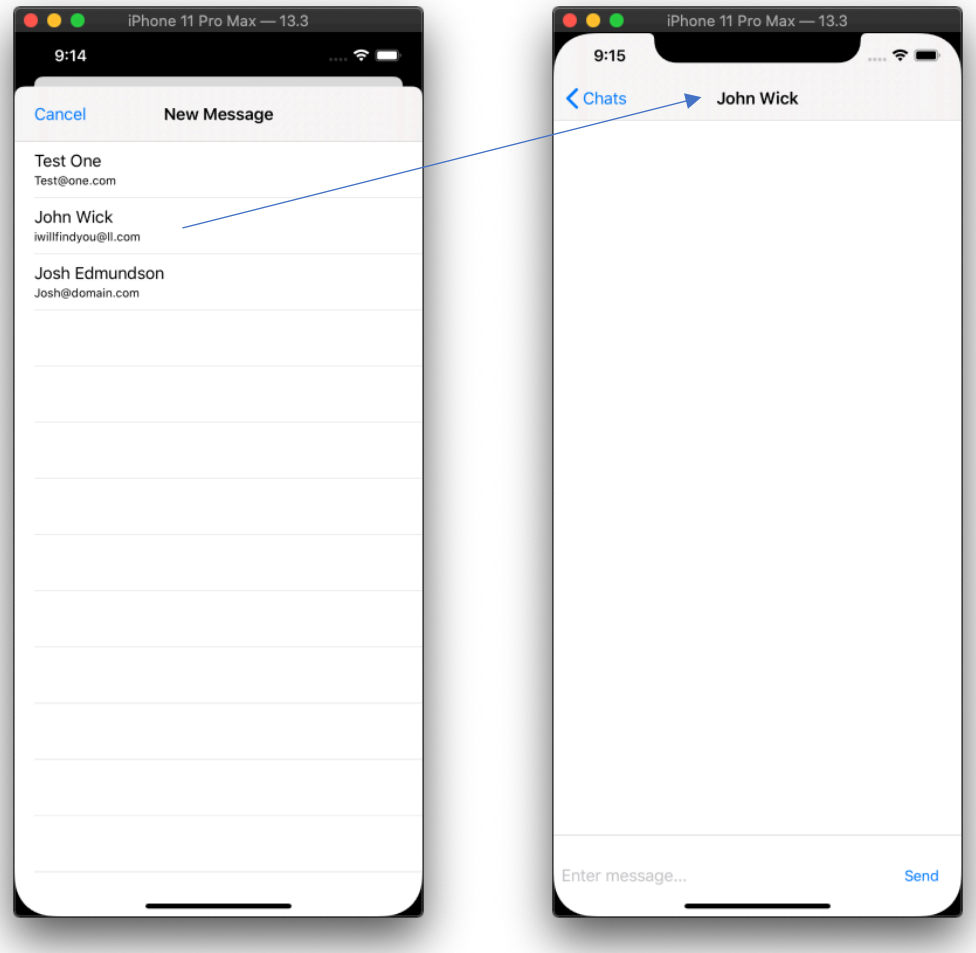
```
30      //Function that will display chatLogController when called.
31      func displayChatLogControllerForUser(user: User) {
32          let chatLogController = ChatLogController(collectionViewLayout: UICollectionViewFlowLayout())
33          chatLogController.hidesBottomBarWhenPushed = true
34          //Sets the attribute user of chatLogController to the argument user.
35          chatLogController.user = user
36          navigationController?.pushViewController(chatLogController, animated: true)
37      }
```

The function now takes the argument passed to it, and sets it as the value of the ChatLogController class' attribute "user".

I've then actually given the class ChatLogController an attribute called "user" of type "User". As soon as this variable is given a value, the title of the chatLogController is changed to the user's name:

```
15      //Sets the title of the instance of ChatLogController to the selected user's name
16      //as soon as the user's name is set.
17      var user: User? {
18          didSet {
19              navigationItem.title = user?.name
20          }
21      }
```

So now if I select a user…



Now I need to work receiving messages.

Modifying the message structure

One of the first issues I have to deal with is the fact these messages are not getting sent to anyone in particular, rather, they are simply being uploaded to an online database. The first thing I've done is specify in each message who the recipient is.

Within the User class, I've added an additional attribute called id:

```
import UIKit

class User: NSObject {

    var id: String?
    var name: String?
    var email: String?

}
```

We will be able to reference each user's unique Firebase ID and store it in this attribute for later use.

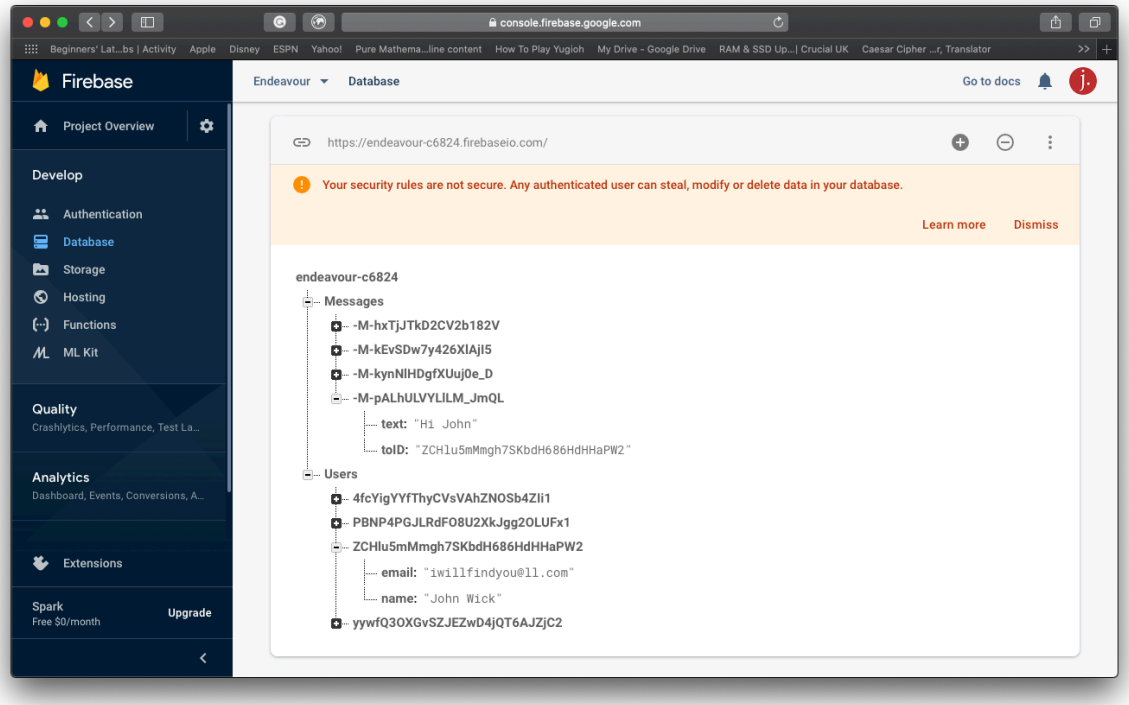
In NewMessageScreen.swift, I've modified the the function fetchUsers() so that when each user's information is fetched and stored in an instance of the User class, their unique Firebase ID is passed to the ID attribute of class User. Now each instance of User found with in the list users has a name, email and ID.

```
//When run, fetches users from firebase
func fetchUsers() {

    //Create a Firebase reference
    Database.database().reference().child("Users").observe(.childAdded, with: { (snapshot) in

        //Creates a variable "dictionary" and stores the values contained within the snapshot as a
        //key-value pair within the dictionary data structure.
        if let dictionary = snapshot.value as? [String: AnyObject] {
            //Creates an instance of User
            let user = User()

            //Gives the attributes of user values
            user.name = dictionary["name"] as? String
            user.email = dictionary["email"] as? String
            user.id = snapshot.key

            //Adds the new user to the list.
            self.users.append(user)

            self.tableView.reloadData()
        }


    }, withCancel: nil)

}
```

I've then modified the handleSend() function within the ChatLogController class so that selected user's ID is uploaded along with the message text to Firebase so that, later on, it will be easy to find and pull down relevant messages for each individual user.

```swift
    //Uploads the text from the inputTextField to the cloud when called
    @objc func handleSend() {
        //Creates a reference to a new node in firebase "Messages"
        let ref = Database.database().reference().child("Messages")
        //Creates a unique ID for this particular message node
        let childRef = ref.childByAutoId()
        //Uploads text and recipient from the text field to the database.
        let toID = user!.id
        let values = ["text": inputTextField.text!, "toID": toID]
        childRef.updateChildValues(values as [AnyHashable : Any])
        //Clears the text field
        inputTextField.text = ""
    }
```

So now when I send a message…

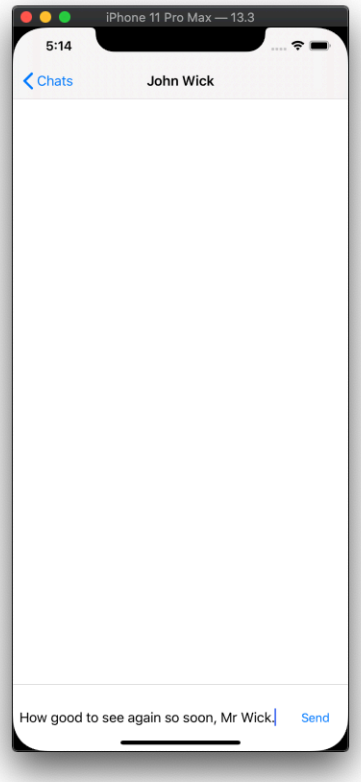


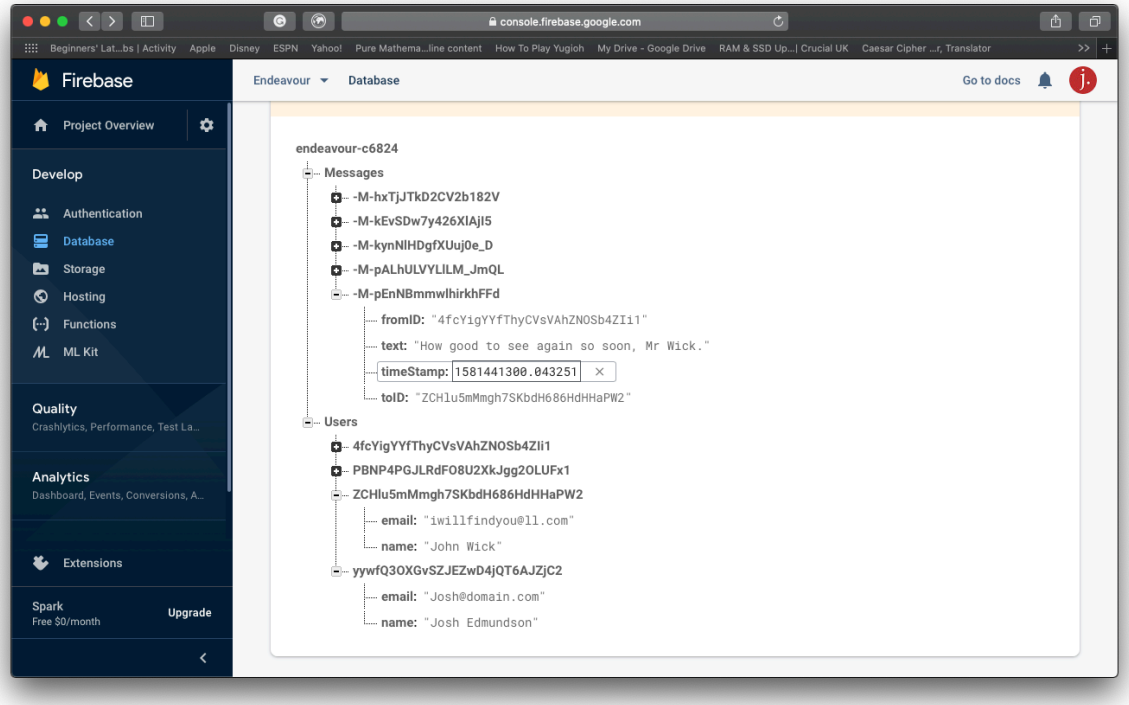
Here you can see when I send a message to John Wick, his unique ID is one of the sub-nodes of the message.

In a similar vein of thought, I've added both "fromID" and "timeStamp" to the message data so that I can later workout who the message is from and when it was sent.

```swift
    //Uploads the text from the inputTextField to the cloud when called
    @objc func handleSend() {
        //Creates a reference to a new node in firebase "Messages"
        let ref = Database.database().reference().child("Messages")
        //Creates a unique ID for this particular message node
        let childRef = ref.childByAutoId()
        //Uploads text and recipient from the text field to the database.
        let toID = user!.id
        let fromID = Auth.auth().currentUser?.uid
        let timeStap = NSDate().timeIntervalSince1970
        let values = ["text": inputTextField.text!, "toID": toID!, "fromID": fromID!, "timeStamp": timeStap] as [String : Any]
        childRef.updateChildValues(values as [AnyHashable : Any])
        //Clears the text field
        inputTextField.text = ""
    }
```

If I send another message:





Displaying messages

One of the issues with the current program is that I can't actually see any of the messages I've sent or received.

In ChatScreen.swift, I've added a new function called "observeMessages" that I will use to gather the messages from Firebase so they can then be displayed to the screen.

```
33      //Fetches the messages from Firebase to be displayed
34      func observeMessages() {
35          let ref = Database.database().reference().child("Messages")
36          ref.observe(.childAdded, with: {(snapshot) in
37
38              print(snapshot)
39
40          }, withCancel: nil)
41      }
```

I've placed a print statement within the scope of the function for the time being so I can check that it works.

I've called the observeMessages() within the viewDidLoad() method. When I build code and run the simulation, I get this in the console:

```
Snap (-M0Me8Y9ee5W_Vyjd8Lf) {
    fromID = 4fcYigYYfThyCVsVAhZNOSb4ZIi1;
    text = "Hello John";
    timeStamp = "1582018631.932859";
    toID = ZCHlu5mMmgh7SKbdH686HdHHaPW2;
}
Snap (-M0MeBeegxfw79Sbgrm7) {
    fromID = 4fcYigYYfThyCVsVAhZNOSb4ZIi1;
    text = "Let us handle this like civilized men ";
    timeStamp = "1582018644.700609";
    toID = ZCHlu5mMmgh7SKbdH686HdHHaPW2;
}
```

So I can see the function is successfully gathering the relevant data from Firebase.

Within the folder "Model", I've added a new Swift file containing a new class that I've written to hold each message's data.

```
 9  import UIKit
10
11  class Message: NSObject {
12
13      var fromID: String?
14      var text: String?
15      var timeStamp: NSNumber?
16      var toID: String?
17
18  }
```

I've then modified observeMessages() so that each messages is imported from Firebase and stored in a Message object.

```
33      //Fetches the messages from Firebase to be displayed
34      func observeMessages() {
35          let ref = Database.database().reference().child("Messages")
36          ref.observe(.childAdded, with: {(snapshot) in
37
38              if let dictionary = snapshot.value as? [String: AnyObject] {
39                  let message = Message()
40                  message.setValuesForKeys(dictionary)
41              }
42
43
44              print(snapshot)
45
46          }, withCancel: nil)
47      }
```

When I build and run the code however, I get this:

```
2020-02-18 10:13:59.653875+0000 Endeavour[10733:496145] *** Terminating app due to uncaught
    exception 'NSUnknownKeyException', reason: '[<Endeavour.Message 0x600000f1b400>
    setValue:forUndefinedKey:]: this class is not key value coding-compliant for the key toID.'
*** First throw call stack:
(
    0   CoreFoundation                      0x00000001058c227e __exceptionPreprocess + 350
    1   libobjc.A.dylib                     0x000000010572fb20 objc_exception_throw + 48
    2   CoreFoundation                      0x00000001058c1e49 -[NSException raise] + 9
    3   Foundation                          0x00000001047a7ee3 -[NSObject(NSKeyValueCoding)
        setValue:forKey:] + 325
    4   Foundation                          0x00000001047a92b9 -[NSObject(NSKeyValueCoding)
        setValuesForKeysWithDictionary:] + 270
    5   Endeavour                           0x00000001034b09cc
        $s9Endeavour10ChatScreenC15observeMessagesyyFySo15FIRDataSnapshotCcfU_ + 780
    6   Endeavour                           0x00000001034abb56
        $sSo15FIRDataSnapshotCIegg_ABIeyBy_TR + 70
    7   Endeavour                           0x0000000103544f96 __63-[FIRDatabaseQuery
        observeEventType:withBlock:withCancelBlock:]_block_invoke + 118
    8   Endeavour                           0x000000010352649c __43-[FChildEventRegistration
        fireEvent:queue:]_block_invoke.66 + 124
    9   libdispatch.dylib                   0x000000010805edd4 _dispatch_call_block_and_release +
        12
    10  libdispatch.dylib                   0x000000010805fd48 _dispatch_client_callout + 8
    11  libdispatch.dylib                   0x000000010806dde6 _dispatch_main_queue_callback_4CF
        + 1500
    12  CoreFoundation                      0x0000000105825049
        __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__ + 9
    13  CoreFoundation                      0x000000010581fca9 __CFRunLoopRun + 2329
    14  CoreFoundation                      0x000000010581f066 CFRunLoopRunSpecific + 438
    15  GraphicsServices                    0x00000010f8c9bb0 GSEventRunModal + 65
    16  UIKitCore                           0x0000000109d2ad4d UIApplicationMain + 1621
    17  Endeavour                           0x00000001034a75ab main + 75
    18  libdyld.dylib                       0x00000001080e7c25 start + 1
)
libc++abi.dylib: terminating with uncaught exception of type NSException
```

Explicitly setting each value of the newly-created message object seemed to do the trick:

```
33      //Fetches the messages from Firebase to be displayed
34      func observeMessages() {
35          let ref = Database.database().reference().child("Messages")
36          ref.observe(.childAdded, with: {(snapshot) in
37
38              if let dictionary = snapshot.value as? [String: AnyObject] {
39                  let message = Message()
40
41                  message.fromID = dictionary["fromID"] as! String?
42                  message.toID = dictionary["toID"] as! String?
43                  message.text = dictionary["text"] as! String?
44                  message.timeStamp = dictionary["timeStamp"] as! NSNumber?
45
46                  print(message.text as Any)
47              }
48
49          }, withCancel: nil)
50      }
```

This is the new console output :

```
Optional("Hello John")
Optional("Let us handle this like civilized men ")
```

I've also made a list called messages in which I will store all the message data from Firebase. I've also made ChatScreen a child of UITableViewController to make displaying current chats easier.

```
12  class ChatScreen: UITableViewController {
13
14      var messages = [Message]()
```

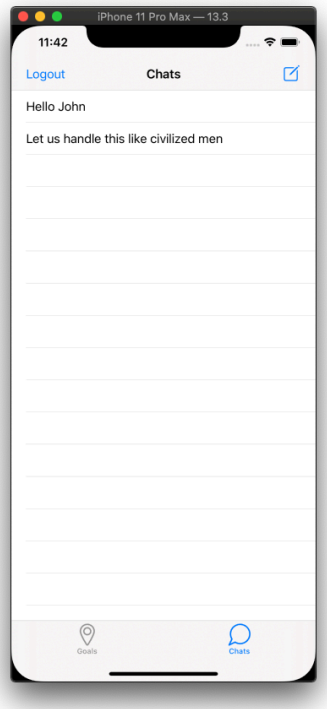
I've the set the number of rows in the table to match the number of items in the messages list and then I've set the title of each cell to that of one of the messages.

```
59      //Sets the number of rows in the table to the number of messages
60      override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
61          return messages.count
62      }
63
64
65      //Sets the title of each cell to the text of the message.
66      override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
67
68          let cell = UITableViewCell(style: .subtitle, reuseIdentifier: "cellID")
69
70          let message = messages[indexPath.row]
71
72          cell.textLabel?.text = message.text
73
74          return cell
75      }
```

After telling the app to reload the table data after fetching it from Firebase:

```
51                  //Reloads the table with the new data
52                  self.tableView.reloadData()
```

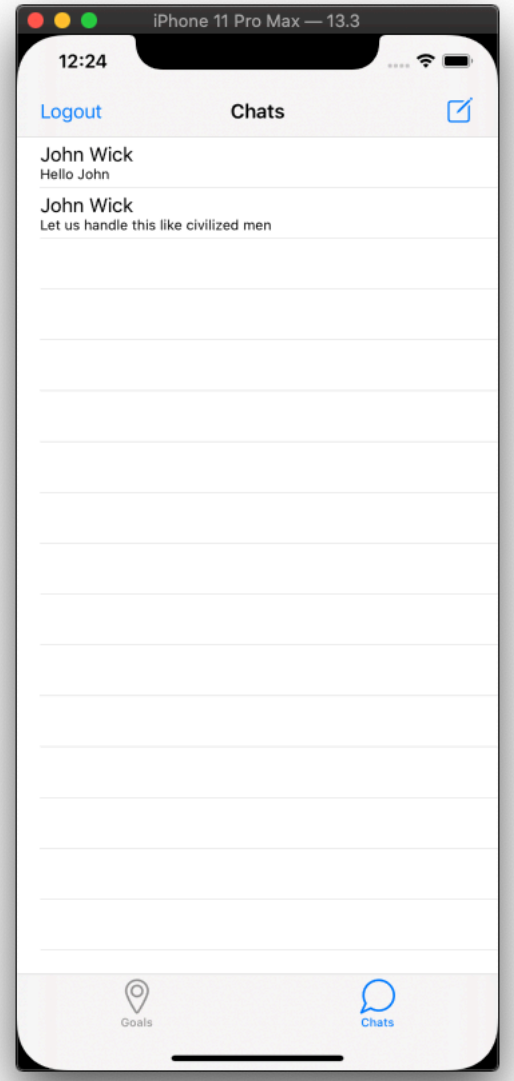
This is what we get:



At the moment, it’s a bit rough and ready, but it’s getting there.

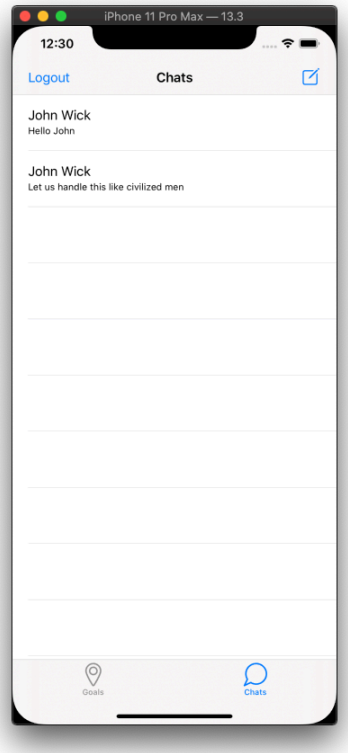
I’ve now added some code that causes each cell in the table to display the name of the user to whom the message was sent, along with the message text.

```
65      //Sets the title of each cell to the text of the message.
66      override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
67
68          //Creates a instance of UITableViewCell
69          let cell = UITableViewCell(style: .subtitle, reuseIdentifier: "cellID")
70
71          //Gets each message from the messages list where the index of the item fetched
72          //matches the row number in the table
73          let message = messages[indexPath.row]
74
75          if let toID = message.toID {
76              //Creates a reference to the message node with a matching toID value.
77              let ref = Database.database().reference().child("Users").child(toID)
78
79              //Takes a snapshot of the values that extend the message ID.
80              ref.observeSingleEvent(of: .value, with: {(snapshot) in
81
82                  if let dictionary = snapshot.value as? [String: AnyObject] {
83
84                      //Sets the value of the cell's text to the value stored at the child node "name".
85                      cell.textLabel?.text = dictionary["name"] as? String
86
87                  }
88
89              })
90          }
91
92          //Sets the subtext of the cell equal to the contents of the message.
93          cell.detailTextLabel?.text = message.text
94
95          return cell
96      }
```

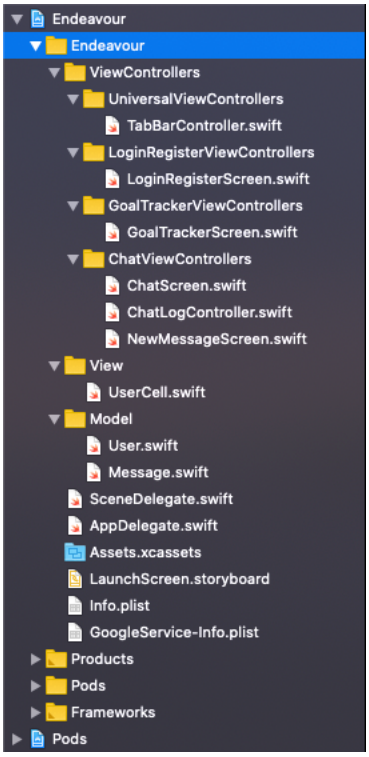


I've then modified the height of the cells in the table to give everything a bit more space:

```
99      //Make the cells in the table bigger
100     override func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
101         return 72
102     }
```



For the sake of organisation, I've tided up some of my code and file organisation:



I've now grouped all my code into specialised folders in an effort to keep track of everything.

I've also created a new group called "View", which contains the UserCell.swift file as you can see in the previous image. I've moved some of the code from the ChatScreen class concerning the customisation of the UITableViewCell objects:

```swift
import UIKit
import Firebase

class UserCell: UITableViewCell {

    var message: Message? {
        didSet {
            if let toID = message?.toID {
                //Creates a reference to the message node with a matching toID value.
                let ref = Database.database().reference().child("Users").child(toID)

                //Takes a snapshot of the values that extend the message ID.
                ref.observeSingleEvent(of: .value, with: {(snapshot) in

                    if let dictionary = snapshot.value as? [String: AnyObject] {

                        //Sets the value of the cell's text to the value stored at the child node "name".
                        self.textLabel?.text = dictionary["name"] as? String

                    }

                })
            }

            //Sets the subtext of the cell equal to the contents of the message.
            self.detailTextLabel?.text = message?.text
        }
    }

    override init(style: UITableViewCell.CellStyle, reuseIdentifier: String?) {
        super.init(style: .subtitle, reuseIdentifier: reuseIdentifier)
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

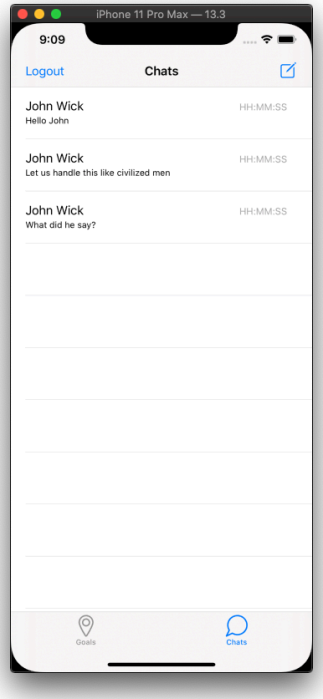
Moving that code to a separate location has drastically reduced the complexity of the override function in the ChatScreen class:

```swift
    //Sets the title of each cell to the text of the message.
    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

        //Creates an instance of the UserCell class for each value of indexPath
        let cell = tableView.dequeueReusableCell(withIdentifier: cellID, for: indexPath) as! UserCell

        //Gets each message from the messages list where the index of the item fetched
        //matches the row number in the table
        let message = messages[indexPath.row]
        //Sets the value of attribute in UserCell to the message constant
        cell.message = message

        return cell
    }
```

When the messages are grouped together, the time stamp of the most recent message will be displayed. I've added some code to the UserCell class to display each message's a time stamp. At the moment, I'm displaying hard-coded text where the time will be shown just to check that the formatting works:

```
39      //Creates a time stap at the edge of the message cell
40      let timeLabel: UILabel = {
41          let label = UILabel()
42          label.text = "HH:MM:SS"
43          label.font = UIFont.systemFont(ofSize: 13)
44          label.textColor = UIColor.lightGray
45          label.translatesAutoresizingMaskIntoConstraints = false
46          return label
47      }()
48
49
50      //Changes the default style of the cell.
51      override init(style: UITableViewCell.CellStyle, reuseIdentifier: String?) {
52          super.init(style: .subtitle, reuseIdentifier: reuseIdentifier)
53          addSubview(timeLabel)
54
55          //Constraints for timeLabel
56          timeLabel.rightAnchor.constraint(equalTo: self.rightAnchor).isActive = true
57          timeLabel.topAnchor.constraint(equalTo: self.topAnchor, constant: 20).isActive = true
58          timeLabel.centerYAnchor.constraint(equalTo: textLabel!.centerYAnchor).isActive = true
59          timeLabel.widthAnchor.constraint(equalToConstant: 100).isActive = true
60          timeLabel.heightAnchor.constraint(equalTo: textLabel!.heightAnchor).isActive = true
61      }
```

Which now outputs the following:

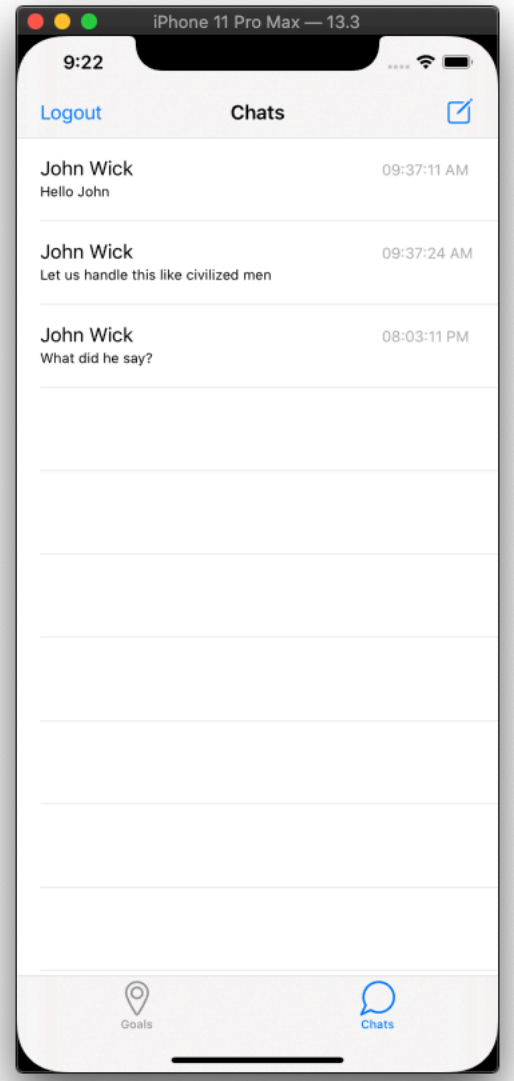


Now I need to display the actual time the message was received.

I've added the following code to the end of the didSet{} statement for the "message" attribute of the UserCell class:

```
36              //Check that the current message has a time stamp
37              //If it does, format the timestamp to a date and set the text of timeLabel
38              //equal to the date.
39              if let seconds = message?.timeStamp?.doubleValue {
40                  let timeStampDate = NSDate(timeIntervalSince1970: seconds)
41
42                  let dateFormatter = DateFormatter()
43                  dateFormatter.dateFormat = "hh:mm:ss a"
44                  timeLabel.text = dateFormatter.string(from: timeStampDate as Date)
```

This turns the time stamp (which is the number of seconds since 1970) into an actual date and then sets the text attribute of timeLabel equal to that date.



In order to group the messages together per user, the first thing I've done is create a new dictionary in the ChatScreen class:

```
15      var messagesDictionary = [String: Message]()
```

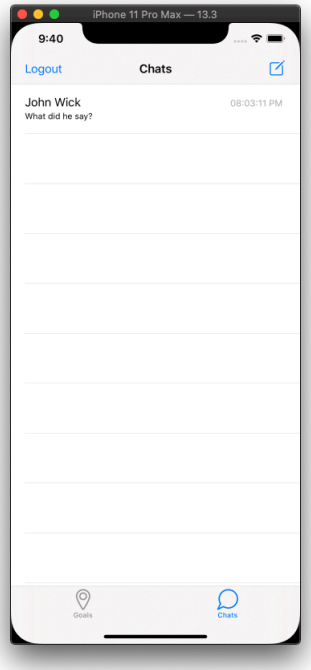
Then, within the observeMessages() function, I've added the highlighted code:

```
39      //Fetches the messages from Firebase to be displayed
40      func observeMessages() {
41          //Create a reference to Firebase
42          let ref = Database.database().reference().child("Messages")
43          //Observe each child attatched to the message node
44          ref.observe(.childAdded, with: {(snapshot) in
45
46              //Stores each message as a Message()
47              if let dictionary = snapshot.value as? [String: AnyObject] {
48                  let message = Message()
49
50                  message.fromID = dictionary["fromID"] as! String?
51                  message.toID = dictionary["toID"] as! String?
52                  message.text = dictionary["text"] as! String?
53                  message.timeStamp = dictionary["timeStamp"] as! NSNumber?
54
55                  self.messages.append(message)
56
57                  //Checks if the message has a toID attribute
58                  //If it does, it creates a key-value pair in the messagesDictionary
59                  //where the ID is the key and the message is the value.
60                  //If the toID already exists in the dictionary, it's value gets updated to the
61                  //text of the latest message sent.
62                  if let toID = message.toID {
63                      self.messagesDictionary[toID] = message
64                  }
65
66                  //Reloads the table with the new data
67                  self.tableView.reloadData()
68              }
69
70          }, withCancel: nil)
71      }
```

This way, I've now got a dictionary with only one entry per user as opposed to per message. The value saved for each user will be the text of the last message sent in that chat.

By adding the following code, I've changed the values in the messages array to now match the values stored in the messagesDictionary. This means only one message will be displayed per user:

```
62                  if let toID = message.toID {
63                      self.messagesDictionary[toID] = message
64
65                      self.messages = Array(self.messagesDictionary.values)
66                  }
```



As you can see, the app now only displays the latest message from John Wick.

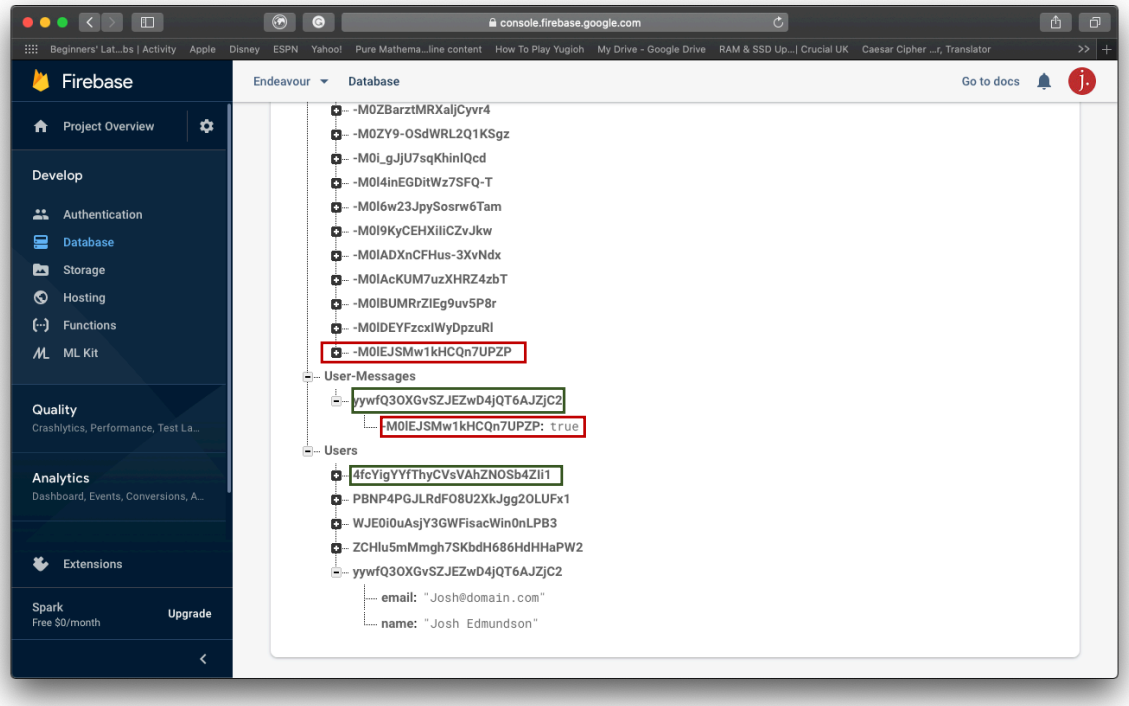
Displaying only the current user's chats

One of the biggest issues with the app at the moment is that it displays all messages sent into the database, regardless of which user is logged in. This, for the sake of privacy, is really not very good.

I've gone and altered the "handleSend()" function within "ChatLogController.swift":

```
99      //Uploads the text from the inputTextField to the cloud when called
100     @objc func handleSend() {
101         //Creates a reference to a new node in firebase "Messages"
102         let ref = Database.database().reference().child("Messages")
103         //Creates a unique ID for this particular message node
104         let childRef = ref.childByAutoId()
105         //Uploads text and recipient from the text field to the database.
106         let toID = user!.id!
107         let fromID = Auth.auth().currentUser!.uid
108         let timeStap = NSDate().timeIntervalSince1970
109         let values = ["text": inputTextField.text!, "toID": toID, "fromID": fromID, "timeStamp": timeStap] as [String : Any]
110
111         //Adds the  key-value pairs stored within the dictionary "values" to the "childByAutoId" node
112         //If those values are successfully added, the completion block runs.
113         childRef.updateChildValues(values) { (error, ref) in
114
115             //Catches and prints any errors to the console
116             if error != nil {
117                 print(error as Any)
118             }
119
120             let messageID = childRef.key
121
122             //Sets up a reference to a node called "User-Messages" and creates a chid node
123             //Based on the user's ID
124             let userMessageRef = Database.database().reference().child("User-Messages").child(fromID)
125             //Adds the message ID to user's ID node.
126             userMessageRef.setValue([messageID : true])
127
128         }
129         //Clears the text field
130         inputTextField.text = ""
131     }
```

Now when the user sends a message, the message is not only uploaded to the “Messages” node, the message ID is also stored at the ID of the user who sent in in the “User-Messages” node. If I send a message using the simulator and look at Firebase:



You can see both a reference to the message and to the user who sent it are now stored as sub-nodes of “User-Messages”.

What this will allow me to do is to find the messages sent by the user currently logged in and only upload those.

Heading over to the “ChatScreenController.swift” file, the function currently used to fetch the user’s messages from Firebase, “observeMessages()” looks like this:

```swift
//Fetches the messages from Firebase to be displayed
func observeMessages() {
    //Create a reference to Firebase
    let ref = Database.database().reference().child("Messages")
    //Observe each child attatched to the message node
    ref.observe(.childAdded, with: {(snapshot) in

        //Stores each message as a Message()
        if let dictionary = snapshot.value as? [String: AnyObject] {
            let message = Message()

            message.fromID = dictionary["fromID"] as! String?
            message.toID = dictionary["toID"] as! String?
            message.text = dictionary["text"] as! String?
            message.timeStamp = dictionary["timeStamp"] as! NSNumber?

            //Checks if the message has a toID attribute
            //If it does, it creates a key-value pair in the messagesDictionary
            //where the ID is the key and the message is the value.
            //If the toID already exists in the dictionary, it's value gets updated to the
            //text of the latest message sent.
            if let toID = message.toID {
                self.messagesDictionary[toID] = message

                self.messages = Array(self.messagesDictionary.values)

                self.messages.sort { (message1, message2) -> Bool in
                    return message1.timeStamp!.intValue > message2.timeStamp!.intValue
                }
            }

            //Reloads the table with the new data
            self.tableView.reloadData()
        }

    }, withCancel: nil)
}
```

The issue with this function is that it fetches every message from Firebase regardless of the user currently logged in on that device.

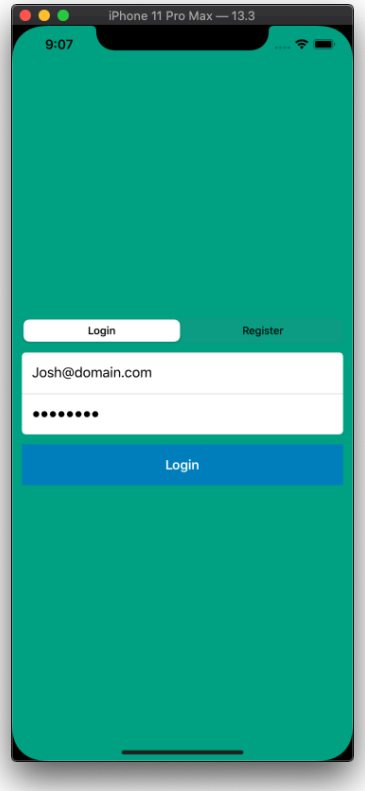
To fix this issue, I've created a new function called "observeUserMessages()" that is essentially a more refined version of "observeMessages()" and called it in "viewDidLoad()":

```swift
//Fetches the messages from Firebase, relevant to the current user, to be displayed.
func observeUserMessages() {
    guard let uid = Auth.auth().currentUser?.uid else {
        return
    }

    let ref = Database.database().reference().child("User-Messages").child(uid)

    ref.observe(.childAdded, with: {(snapshot) in

        let messageID = snapshot.key
        let messagesReference = Database.database().reference().child("Messages").child(messageID)

        messagesReference.observe(.value, with: { (snapshot) in

            //Stores each message as a Message()
            if let dictionary = snapshot.value as? [String: AnyObject] {
                let message = Message()

                message.fromID = dictionary["fromID"] as! String?
                message.toID = dictionary["toID"] as! String?
                message.text = dictionary["text"] as! String?
                message.timeStamp = dictionary["timeStamp"] as! NSNumber?

                //Checks if the message has a toID attribute
                //If it does, it creates a key-value pair in the messagesDictionary
                //where the ID is the key and the message is the value.
                //If the toID already exists in the dictionary, it's value gets updated to the
                //text of the latest message sent.
                if let toID = message.toID {
                    self.messagesDictionary[toID] = message

                    self.messages = Array(self.messagesDictionary.values)

                    //Sorts the different chats so the most recently active is
                    //displayed at the top of the table.
                    self.messages.sort { (message1, message2) -> Bool in
                        return message1.timeStamp!.intValue > message2.timeStamp!.intValue
                    }
                }

                //Reloads the table with the new data
                self.tableView.reloadData()
            }

        }, withCancel: nil)

    }, withCancel: nil)
}
```

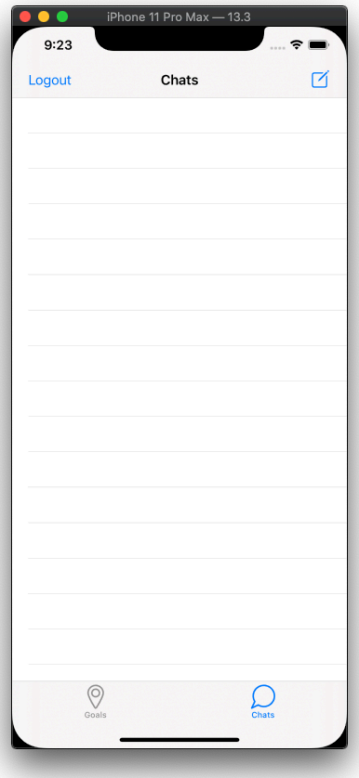
Whereas “observeMessages()” observed all the messages found at the “Messages” node, “observeUserMessages” sets up a reference to the current user’s ID as a sub-node of “UserMessages”. This observation gives the program the message IDs sent by the current user. I then set up another reference to the database, this time to the sub-node of “Messages” that matches the message IDs of the messages sent by the current user. The program then retrieves the relevant message data and stores them in the “messagesDictionary” where they can be retrieved for display.

After deleting all the messages currently in the database, let’s look at the simulator:

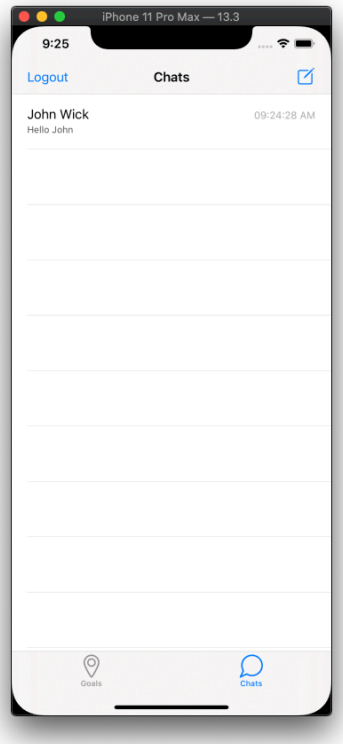
First, I'll log into my account,



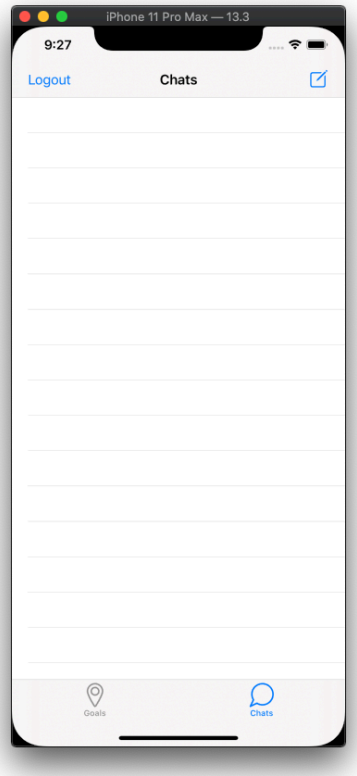
Let's take a look at the "Chats" screen:



It’s currently empty as there are no messages stored in the database. I’ll send one from my account:



Now, if I log out of josh@domain.com and log into the “Test One” account:



We see that chats is empty.

For some reason, when I tried to log out of josh@domain.com, I got this error in the console:

```
2020-02-24 09:28:05.381300+0000 Endeavour[1294:80878] 6.15.0 - [Firebase/Database][I-RDB038012] Listener at
    /User-Messages/PBNP4PGJLRdFO8U2XkJgg2OLUFx1 failed: permission_denied
```

I had to restart the simulator to actually register the logout…

This is a bug I will fix later as it is not a pressing matter for the moment.

Something else I need to fix is the fact that, currently, the recipient does not receive the messages sent to them. I've modified the code for "handleSend()" to fix this issue:

```
99      //Uploads the text from the inputTextField to the cloud when called
100     @objc func handleSend() {
101         //Creates a reference to a new node in firebase "Messages"
102         let ref = Database.database().reference().child("Messages")
103         //Creates a unique ID for this particular message node
104         let childRef = ref.childByAutoId()
105         //Uploads text and recipient from the text field to the database.
106         let toID = user!.id!
107         let fromID = Auth.auth().currentUser!.uid
108         let timeStap = NSDate().timeIntervalSince1970
109         let values = ["text": inputTextField.text!, "toID": toID, "fromID": fromID, "timeStamp": timeStap] as [String : Any]
110
111         //Adds the  key-value pairs stored within the dictionary "values" to the "childByAutoId" node
112         //If those values are successfully added, the completion block runs.
113         childRef.updateChildValues(values) { (error, ref) in
114
115             //Catches and prints any errors to the console
116             if error != nil {
117                 print(error as Any)
118             }
119
120             guard let messageID = childRef.key else {
121                 return
122             }
123
124             //Sets up a reference to a node called "User-Messages" and creates a chid node
125             //Based on the user's ID
126             let userMessageRef = Database.database().reference().child("User-Messages").child(fromID)
127             //Adds the message ID to user's ID node.
128             userMessageRef.updateChildValues([messageID: true]) { (error, ref) in
129                 print("Inside userMessageRef")
130                 if error != nil{
131                     print(error as Any)
132                     return
133                 }
134             }
135
136             //Creates a node in "User-Messages" based on the recipients ID
137             //Stores the message ID at that node.
138             let recipientUserMessagesRef = Database.database().reference().child("User-Messages").child(toID)
139             recipientUserMessagesRef.updateChildValues([messageID: true]) {(error, ref) in
140                 if error != nil {
141                     print(error as Any)
142                     return
143                 }
144             }
145
146         }
147         //Clears the text field
148         inputTextField.text = ""
149     }
```

If you look at line 138, the block of code you'll find there, when run, adds the message ID to the recipient user's sub-node of "User-Messages". When I switch accounts to theirs, the app will now load those message references as well.

On line 128, I've changed the method called on "userMessageRef" from ".setValue()" to ".updateChildValues". I discovered that ".setValue()" literally sets a single value for the node

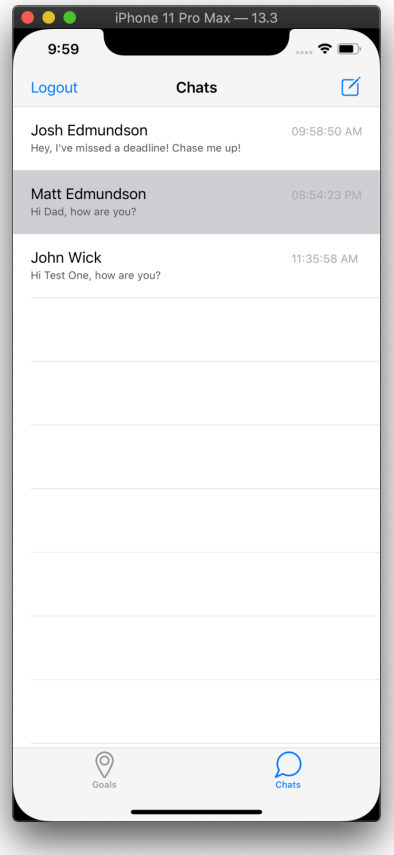
you call it on, so even though I had sent multiple messages to multiple people the database only stored the latest of those messages. In order for this to work however, I've had to use a "guard" statement on line 120. This is because, from what I understand, ".updateChildValues()" does not work if you pass it an optional value. By running "messageID" through a "guard" statement, I unwrap the optional value, allowing it to be used by ".updateChildValues()".

Another bug I've come across, that I will again have to resolve later, is logging in and out doesn't refresh the table view. This means that, if I log out of my account and then log into John Wick's account, all the chats from my account will be displayed. The only way to get around this is to log out, re-run the simulator, and then log in.

Selecting an existing chat

Currently you can only see the latest message displayed on the chat application. For a fully functioning chat app, this is probably something I should fix.

The first issue is the when I select an existing chat cell in the "Chats" screen, nothing happens:



To sort this out, I've refactored some code, modifying the "Messages" class and the "UserCell" class.

```swift
import UIKit
import Firebase

class Message: NSObject {

    var fromID: String?
    var text: String?
    var timeStamp: NSNumber?
    var toID: String?


    func chatPartnerID() -> String? {

        return fromID == Auth.auth().currentUser?.uid ? toID : fromID

    }

}
```

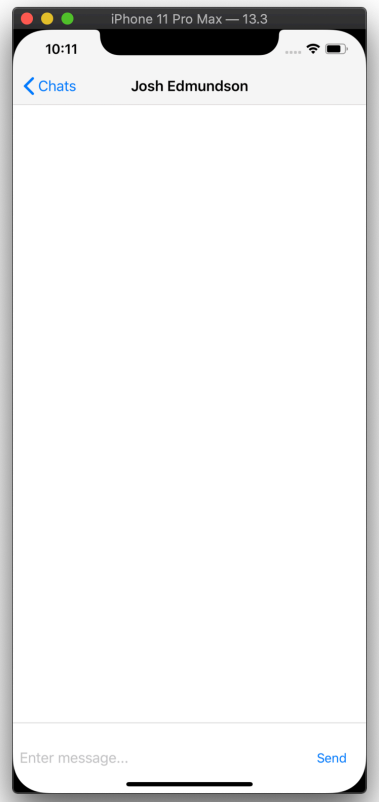
I've added the "chatPartnerID()" function which, when called, checks whether the message it is called on has been sent by the current user or not, and then returns that information. This code was previously in the "UserCell" class.

Within the "ChatScreen.swift" file, I've also added a new function:

```swift
    //Displays the given user's chat screen when their cell is selected in the table view
    override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {

        //Find the message corresponding to the cell selected
        let message = messages[indexPath.row]

        //Get's the ID of the other user in the chat by calling the "chatPartnerID" method.
        guard let chatPartnerID = message.chatPartnerID() else {
            return
        }

        //Sets up a reference to the aforementioned user's node
        let ref = Database.database().reference().child("Users").child(chatPartnerID)

        //Observes the values at the user's node
        ref.observeSingleEvent(of: .value, with: { (snapshot) in

            //Creates a dictionary to store the observed values
            guard let dictionary = snapshot.value as? [String: AnyObject] else {
                return
            }

            //Creates a user object
            let user = User()

            //Sets the attribute of the object based on the data stored in the dictionary
            user.name = dictionary["name"] as? String
            user.email = dictionary["email"] as? String
            user.id = snapshot.key

            //Display's the "chatLogContoller" for the selected user.
            self.displayChatLogControllerForUser(user: user)

        }, withCancel: nil)
    }
```

This overrides the function that previously told the app to do nothing when a pre-existing chat cell was selected. Instead, when a cell is selected, the message data for the corresponding cell is retrieved from the “messages” array. This information is used to workout which users are in that chat and which user is not the one currently logged in on that device. The corresponding chat screen is displayed using the “displayChatLogControllerForUser()” function.

Now if I select a cell in the simulator:



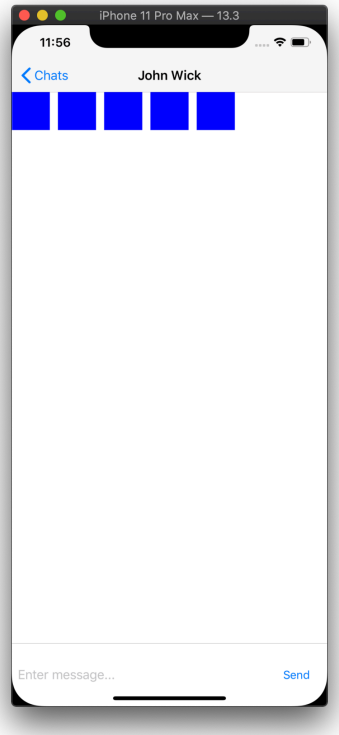
It takes me to the corresponding chat.

Displaying all the user’s messages in a chat

When you to message a user, you can’t see any of the messages either of you had previously sent. The “ChatLogController” class responsible for each individual messaging screen is a child of “UICollectionViewController” which means I can format it in a way very similar to a “UITableView” by adding cells. I will then use these cells to hold each user’s messages.

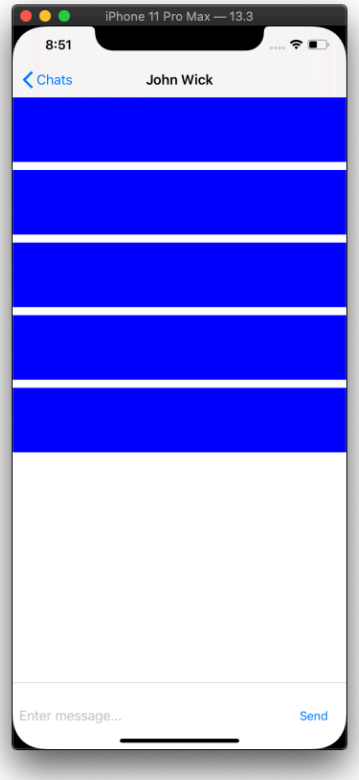
```
34      //Create a cell ID
35      let cellID = "cellID"
36
37      override func viewDidLoad() {
38          super.viewDidLoad()
39
40          collectionView?.backgroundColor = .white
41          collectionView?.register(UICollectionViewCell.self, forCellWithReuseIdentifier:
                cellID)
42
43          setupInputComponents()
44      }
45
46
47      //Sets the number of cells in the collection view
48      override func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection
            section: Int) -> Int {
49          return 5
50      }
51
52
53      //Controlls the content of each cell
54      override func collectionView(_ collectionView: UICollectionView, cellForItemAt
            indexPath: IndexPath) -> UICollectionViewCell {
55          let cell = collectionView.dequeueReusableCell(withReuseIdentifier: cellID, for:
                indexPath)
56
57          cell.backgroundColor = .blue
58
59          return cell
60      }
```

With the above code, I've specified the number of cells in the view (for now, arbitrarily 5) and the colour of each cell. I've then registered the cells with the current class. When I run the simulator:



I’ve added in an extra bit of code to make the blue cells stretch across the whole screen:

```
47      //Set the size of each cell being displayed
48      func collectionView(_ collectionView: UICollectionView, layout collectionViewLayout: UICollectionViewLayout, sizeForItemAt indexPath: IndexPath) -> CGSize {
49          return CGSize(width: view.frame.height, height: 80)
50      }
```



The issue to fix now is retrieving the actual message text.

The first thing I’ve done is create a function in “ChatLogController.swift” that will retrieve the messages either sent or received by the current user:

```
    //Observes the messages of which the current user is either the sender or recipient and appends them to the
    //"messages" list.
    func observeMessages() {

        //Unwrapps the current user UID
        guard let uid = Auth.auth().currentUser?.uid else {
            return
        }

        //Sets up a reference to the "User-Messages" node in Firebase
        let userMessagesRef = Database.database().reference().child("User-Messages").child(uid)

        //Observes each message at the current user's UID sub-node
        userMessagesRef.observe(.childAdded, with: { (snapshot) in

            //Gets the message ID as the snapshot key
            let messageID = snapshot.key

            //Sets up another reference to the "Messages" node this time
            let messagesRef = Database.database().reference().child("Messages").child(messageID)

            //Observes each message either sent or recieved by the current user
            messagesRef.observe(.value, with: { (snapshot) in

                //Unwrapps the values found and stores them in a dictionary
                guard let dictionary = snapshot.value as? [String: AnyObject] else {
                    return
                }

                //Creates a message object for each message found
                let message = Message()

                //Sets the "message" attributes to the corresponding values from the database
                message.text = dictionary["text"] as? String
                message.fromID = dictionary["fromID"] as? String
                message.timeStamp = dictionary["timeStamp"] as? NSNumber
                message.toID = dictionary["toID"] as? String

                //Adds the each "message" to the "messages" array
                self.messages.append(message)

                //Reloads the table data to display all the messages.
                self.collectionView.reloadData()

            }, withCancel: nil)

        }, withCancel: nil)

    }
```

It stores these messages as “Message” objects in an array called “messages”. I’ve then called this function in the “didSet” method for the “user” attribute of “ChatLogController”:

```
    //Sets the title of the instance of ChatLogController to the selected user's name
    //as soon as the user's name is set.
    var user: User? {
        didSet {
            navigationItem.title = user?.name

            //Calls observe messages
            observeMessages()
        }
    }
```

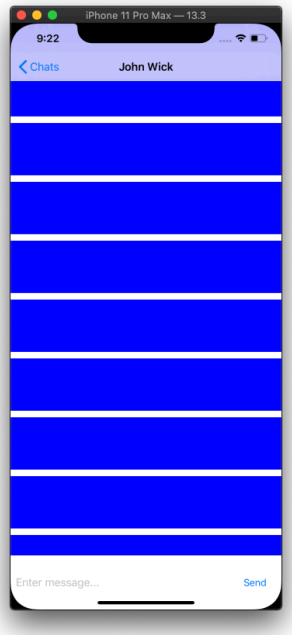
The next thing I did was modify the number of cells being displayed to match the number of items in the “messages” array:

```
    //Sets the number of cells in the collection view
    override func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        //Sets the number of cells to be displayed equal to the number of messages in "messages".
        return messages.count
    }
```

I then also gave the UIView at the bottom of the screen a solid white colour so that you can't see the message boxes behind it:

```
95      override func viewDidLoad() {
96          super.viewDidLoad()
97
98          //Sets the background colour of the UIView at the bottom of the screen to solid white
99          //This makes sure you cannot see the messages through the text bar.
100         collectionView?.backgroundColor = .white
101         collectionView?.register(UICollectionViewCell.self, forCellWithReuseIdentifier: cellID)
102
103         setupInputComponents()
104     }
```

Now the simulator gives me this:



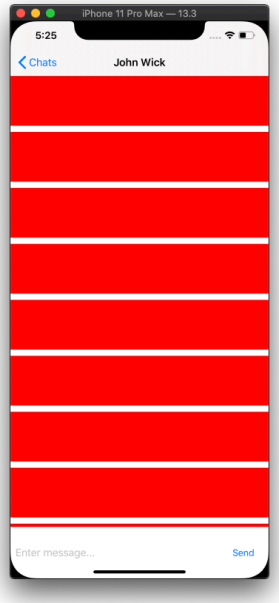
I'm currently displaying the blue boxes as cells within a "UICollectionView" subclass that I've made (ChatLogController). Collection views work slightly differently to table views as their cells don't have a default text property. I've therefore made a new class in the "View" folder called "ChatMessageCell" which inherits from "UICollectionViewCell" that I will be able to manipulate and customise to a much greater degree than a default "UICollectionViewCell" object.

```swift
import UIKit

class ChatMessageCell: UICollectionViewCell {

    //Creates an attribute "textView" for the class that will allow me to display message text
    let textView: UITextView = {
        let tv = UITextView()
        tv.text = "Sample text for now"
        tv.font = UIFont.systemFont(ofSize: 16)
        return tv
    }()

    //Override the constructor function
    override init(frame: CGRect) {
        //Inherit "frame" from the parent class
        super.init(frame: frame)

        //Set the background colour to red.
        backgroundColor = .red
    }

    //This is an extra peice of code needed to make the above "override init" work.
    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

}
```

I've then registered this class within the "ChatLogController" class so the collection view knows to display these custom cells.

```swift
collectionView?.register(ChatMessageCell.self, forCellWithReuseIdentifier: cellID)
```

I've then also removed the line of code that made the cells appear blue. Notice, I've set the background colour to red in the custom cell class just so that I can check it works.



The cells are displayed with a red background, so everything has connected successfully.

I've modified the "ChatMessageCell" class so that now the "textView" actually gets displayed for each cell (I've also removed the red background):

```swift
import UIKit

class ChatMessageCell: UICollectionViewCell {

    //Creates an attribute "textView" for the class that will allow me to display message text
    let textView: UITextView = {
        let tv = UITextView()
        tv.text = "Sample text for now"
        tv.font = UIFont.systemFont(ofSize: 16)
        tv.translatesAutoresizingMaskIntoConstraints = false
        return tv
    }()

    //Override the constructor function
    override init(frame: CGRect) {
        //Inherit "frame" from the parent class
        super.init(frame: frame)

        addSubview(textView)

        //Set up the constraints for "textView"
        textView.rightAnchor.constraint(equalTo: self.rightAnchor).isActive = true
        textView.topAnchor.constraint(equalTo: self.topAnchor).isActive = true
        textView.widthAnchor.constraint(equalToConstant: 200).isActive = true
        textView.heightAnchor.constraint(equalTo: self.heightAnchor).isActive = true

    }

    //This is an extra peice of code needed to make the above "override init" work.
    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

}
```

I've also had to modify a line of code in the "ChatLogController" class as I had incorrectly set the width of each cell to be equal to its height in one of the "override func collectionView(…)" statements. When I tried running the code before making this change, I kept getting width constraint errors as the width I defined in the "ChatMessageCell" class was out-of-bounds.

Before:

```swift
    //Set the size of each cell being displayed
    func collectionView(_ collectionView: UICollectionView, layout collectionViewLayout: UICollectionViewLayout, sizeForItemAt indexPath: IndexPath) -> CGSize {
        return CGSize(width: view.frame.height, height: 80)
    }
```

After:

```swift
    //Set the size of each cell being displayed
    func collectionView(_ collectionView: UICollectionView, layout collectionViewLayout: UICollectionViewLayout, sizeForItemAt indexPath: IndexPath) -> CGSize {
        return CGSize(width: view.frame.width, height: 80)
    }
```

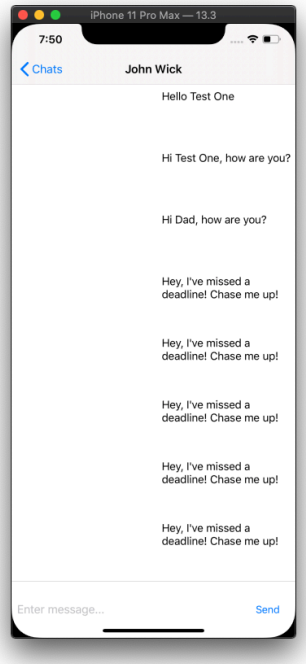
Running the code:

I've then replaced the dummy text show above with the text from each message stored in the "messages" list by modifying once of the "override func collectionView" statements:

```
120     //Controlls the content of each cell
121     override func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
122
123         //Creates a cell as an instance of "ChatMessageCell"
124         let cell = collectionView.dequeueReusableCell(withReuseIdentifier: cellID, for: indexPath) as! ChatMessageCell
125
126         //Retrieves the text of each message in the "messages" list
127         let message = messages[indexPath.item]
128
129         //Sets the "text" attribute of each cell equal to the text of one of the messages.
130         cell.textView.text = message.text
131
132         return cell
133     }
```

Now, each cell is created as an instance of the "ChatMessageCell" class and has its "text" attribute set to be equal to the text of one of the messages.

Running the simulator:



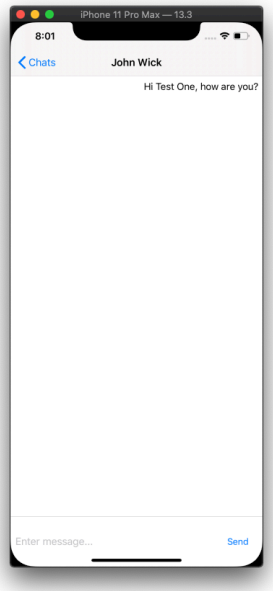
There are three issues running the simulator just now has revealed:

1. Some of the texts above don't belong in this chat.
2. Some of the texts have been received, not sent, yet they are all formatted the same way.
3. It seems as though the "Hey, I've missed a deadline! Chase me up!" message is sent every time that user loads up the goal screen on their application. This is not necessarily a bad thing as it will annoy their peer into responding, but it was not intentional and thus is something to keep in mind.

I've dealt with the first issue by making sure only the correct messages for this chat are displayed.

```
//Makes sure only messages that belong in that chat are displayed
if message.chatPartnerID() == self.user?.id {
    //Adds the each "message" to the "messages" array
    self.messages.append(message)

    //Reloads the table data to display all the messages.
    self.collectionView.reloadData()
}
```

(The above code is within the "observeMessages()" method of "ChatLogController"). Now each message in "messages" is checked to make sure it belongs in that group chat.



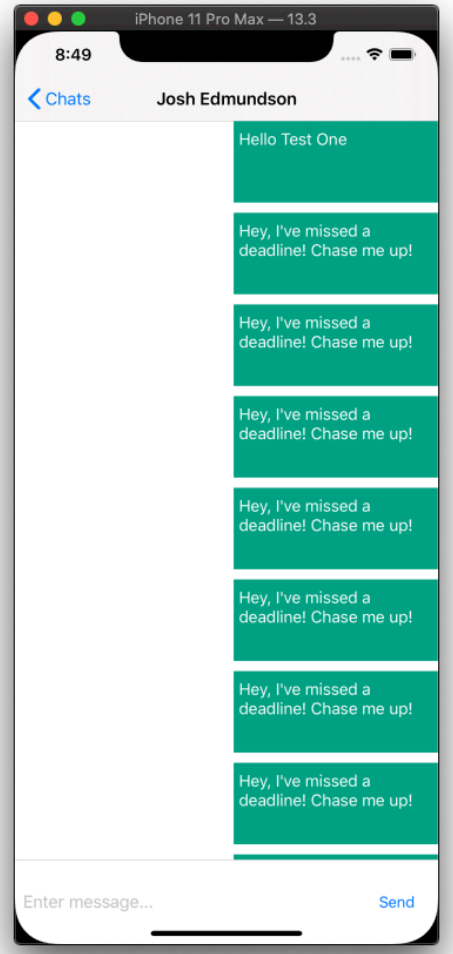
As you can see, this has had quite a dramatic impact on the message feature.

The next step will be to add text bubbles.

The first thing I've done is create a new attribute in the "ChatMessageCell" class called "bubbleView" and set up it's constraints:

```swift
import UIKit

class ChatMessageCell: UICollectionViewCell {

    //Creates an attribute "textView" for the class that will allow me to display message text
    let textView: UITextView = {
        let tv = UITextView()
        tv.text = "Sample text for now"
        tv.font = UIFont.systemFont(ofSize: 16)
        tv.translatesAutoresizingMaskIntoConstraints = false
        tv.backgroundColor = .clear
        tv.textColor = .white
        return tv
    }()

    //Create the background text bubble for each message.
    let bubbleView: UIView = {
        let view = UIView()
        view.backgroundColor = UIColor(displayP3Red: 22/255, green: 160/255, blue: 133/255, alpha: 1.0)
        view.translatesAutoresizingMaskIntoConstraints = false
        return view
    }()


    //Override the constructor function
    override init(frame: CGRect) {
        //Inherit "frame" from the parent class
        super.init(frame: frame)

        addSubview(bubbleView)
        addSubview(textView)

        //Set up constraints for the "bubbleView"
        bubbleView.rightAnchor.constraint(equalTo: self.rightAnchor).isActive = true
        bubbleView.topAnchor.constraint(equalTo: self.topAnchor).isActive = true
        bubbleView.widthAnchor.constraint(equalToConstant: 200).isActive = true
        bubbleView.heightAnchor.constraint(equalTo: self.heightAnchor).isActive = true

        //Set up the constraints for "textView"
        textView.rightAnchor.constraint(equalTo: self.rightAnchor).isActive = true
        textView.topAnchor.constraint(equalTo: self.topAnchor).isActive = true
        textView.widthAnchor.constraint(equalToConstant: 200).isActive = true
        textView.heightAnchor.constraint(equalTo: self.heightAnchor).isActive = true

    }

    //This is an extra peice of code needed to make the above "override init" work.
    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

}
```

First, I've given the "bubbleView" a background colour and then I've added it as a subview of the main view. I've then defined it's constraints to mirror those of the text view. I've also modified some of the properties of the "textView" attribute by making it transparent and giving it a text colour of white. This way you will be able to see the "bubble" behind each text message:



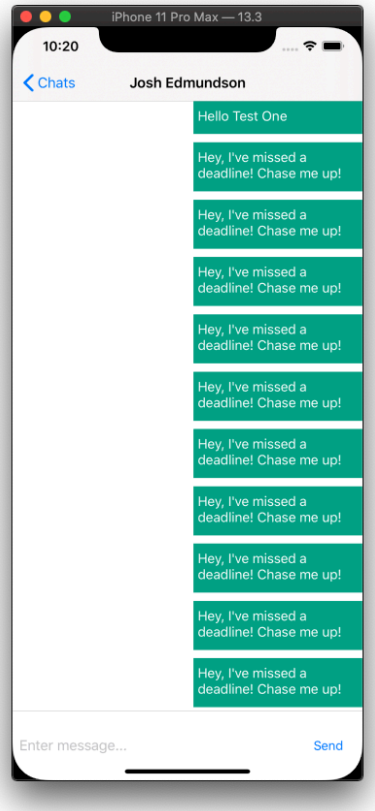
A bit rough and ready but it's a start.

The first thing I've taken care of is the excess space underneath the text of each message.

Within "ChatLogController", I've added the following code:

```
111     //Set the size of each cell being displayed
112     func collectionView(_ collectionView: UICollectionView, layout collectionViewLayout: UICollectionViewLayout, sizeForItemAt indexPath: IndexPath) -> CGSize {
113
114         //Creates a height variable for the cell and gives it a base value
115         var height: CGFloat = 80
116
117         //Checks the message has text
118         if let text = messages[indexPath.item].text {
119             height = estimateFrameForText(text: text).height + 20
120         }
121
122         return CGSize(width: view.frame.width, height: height)
123     }
124
125
126     //Gets an estimate for the size of a message cell based off of the text to be displayed.
127     private func estimateFrameForText(text: String) -> CGRect {
128
129         let size = CGSize(width: 200, height: 1000)
130         let options = NSStringDrawingOptions.usesFontLeading.union(.usesLineFragmentOrigin)
131
132         return NSString(string: text).boundingRect(with: size, options: options, attributes: [NSAttributedString.Key.font : UIFont.systemFont(ofSize: 16)], context: nil)
133     }
```

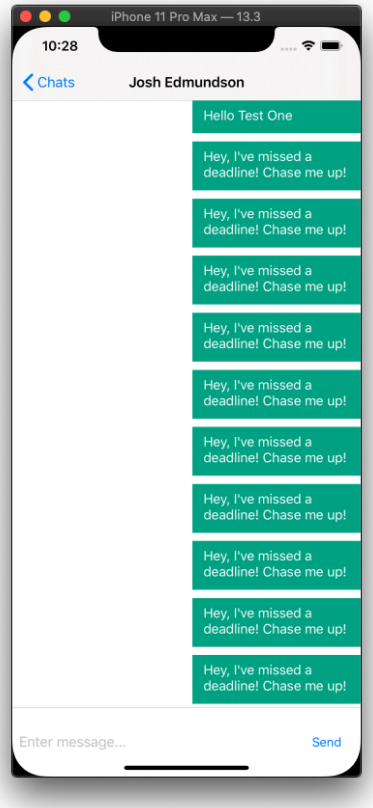
This sets the height of the messageBubble based on the number of lines of text in the text message.



You can see here we've managed to get rid of the excess space on the underside of each message. Now I'll add some more padding to the sides.

I've modified the left and right anchor constraints of the "textView" in "ChatMessageCell.swift":

```
47          //Set up the constraints for "textView"
48          textView.leftAnchor.constraint(equalTo: bubbleView.leftAnchor, constant: 8).isActive = true
49          textView.topAnchor.constraint(equalTo: self.topAnchor).isActive = true
50          textView.rightAnchor.constraint(equalTo: bubbleView.rightAnchor).isActive = true
51          textView.heightAnchor.constraint(equalTo: self.heightAnchor).isActive = true
```



Which has given the text a little more breathing room on either side.

I've then modified the code of the "ChatMessageCell" class:

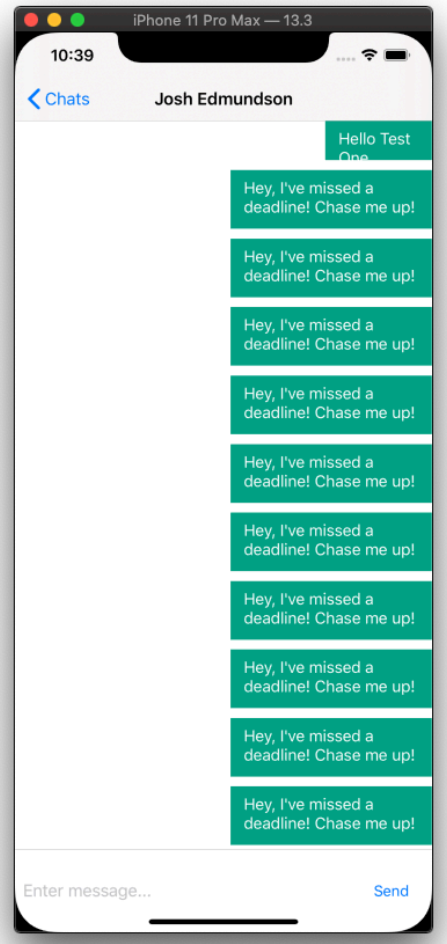
```
33      //Set up a variable that will allow me to access and modify the bubbleView's width externally
34      var bubbleWidthAnchor: NSLayoutConstraint?
35
36
37      //Override the constructor function
38      override init(frame: CGRect) {
39          //Inherit "frame" from the parent class
40          super.init(frame: frame)
41
42          addSubview(bubbleView)
43          addSubview(textView)
44
45          //Set up constraints for the "bubbleView"
46          bubbleView.rightAnchor.constraint(equalTo: self.rightAnchor).isActive = true
47          bubbleView.topAnchor.constraint(equalTo: self.topAnchor).isActive = true
48          bubbleWidthAnchor = bubbleView.widthAnchor.constraint(equalToConstant: 200)
49          bubbleWidthAnchor?.isActive = true
50          bubbleView.heightAnchor.constraint(equalTo: self.heightAnchor).isActive = true
51
52          //Set up the constraints for "textView"
53          textView.leftAnchor.constraint(equalTo: bubbleView.leftAnchor, constant: 8).isActive = true
54          textView.topAnchor.constraint(equalTo: self.topAnchor).isActive = true
55          textView.rightAnchor.constraint(equalTo: bubbleView.rightAnchor).isActive = true
56          textView.heightAnchor.constraint(equalTo: self.heightAnchor).isActive = true
57
58      }
```

I've created a new variable (line 34) which will allow me to access and modify the message bubble's width externally. I can use this variable to automatically adjust the message bubble width base on the size of the text content.

Within "ChatLogController" I've then modified one of the "override" functions:

```
143     //Controlls the content of each cell
144     override func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
145
146         //Creates a cell as an instance of "ChatMessageCell"
147         let cell = collectionView.dequeueReusableCell(withReuseIdentifier: cellID, for: indexPath) as! ChatMessageCell
148
149         //Retrieves the text of each message in the "messages" list
150         let message = messages[indexPath.item]
151
152         //Sets the "text" attribute of each cell equal to the text of one of the messages.
153         cell.textView.text = message.text
154
155         //Modify the message buble width
156         cell.bubbleWidthAnchor?.constant = estimateFrameForText(text: message.text!).width
157
158         return cell
159     }
```

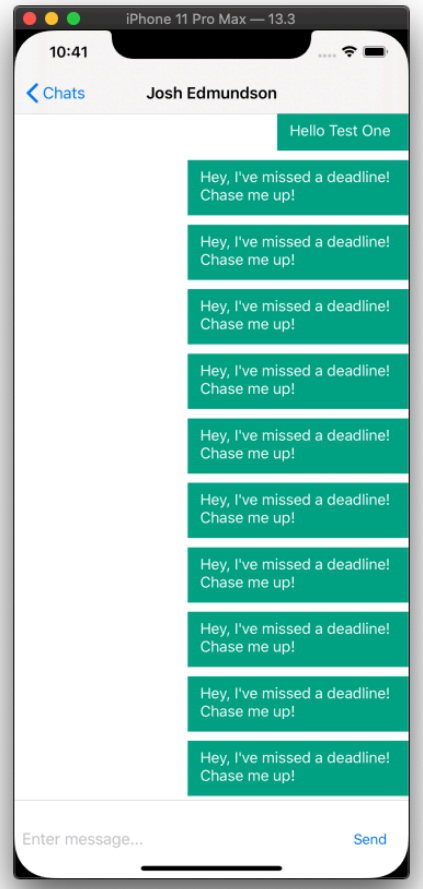
I've used the "estimateFrameForText" function defined earlier to estimate the width of the cell:



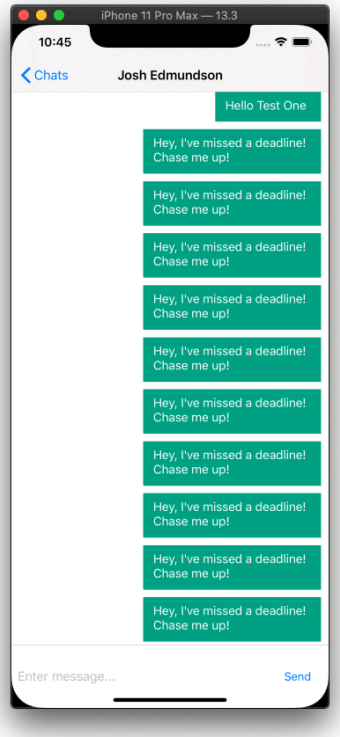
We can see here that the bubble is now adjusting its width based on the message content. Unfortunately, this is also causing some unwanted clipping.

Simply adding a few extra pixels to the width seems to solve that problem:

```
155         //Modify the message buble width
156         cell.bubbleWidthAnchor?.constant = estimateFrameForText(text: message.text!).width + 32
```

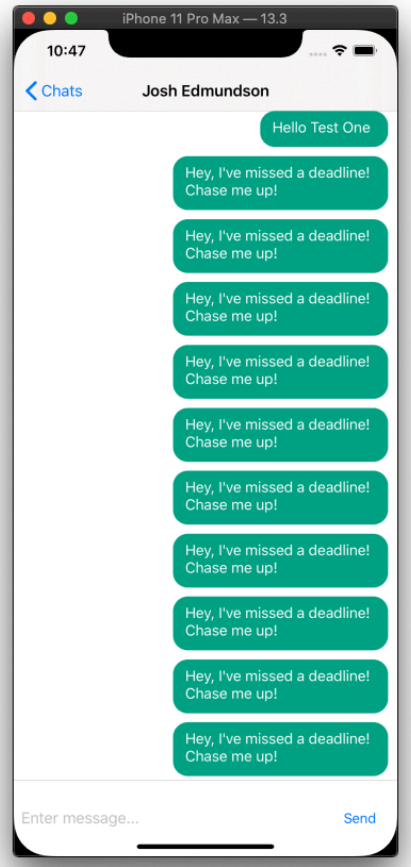


I've then also offset the postion of the "bubbleView"'s right anchor by 10 pixels to the left which gives me this:



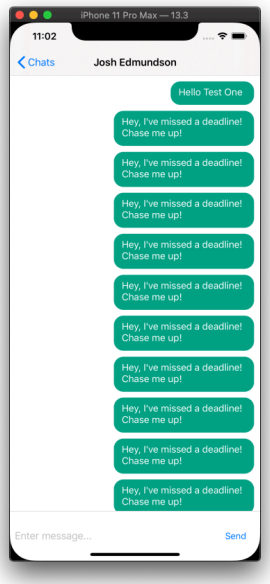
I've then modified the "bubbleView" constant in "ChatMessageCell" to give each bubble rounded corners:

```
24      //Create the background text bubble for each message.
25      let bubbleView: UIView = {
26          let view = UIView()
27          view.backgroundColor = UIColor(displayP3Red: 22/255, green: 160/255, blue: 133/255, alpha: 1.0)
28          view.translatesAutoresizingMaskIntoConstraints = false
29          view.layer.cornerRadius = 16
30          view.layer.masksToBounds = true
31          return view
32      }()
```



To give the top message a bit of padding from the navigation bar, I've added the following to the "viewDidLoad" function of "ChatLogController":

```
collectionView?.contentInset = UIEdgeInsets(top: 10, left: 0, bottom: 58, right: 0)
```



Adding Goals

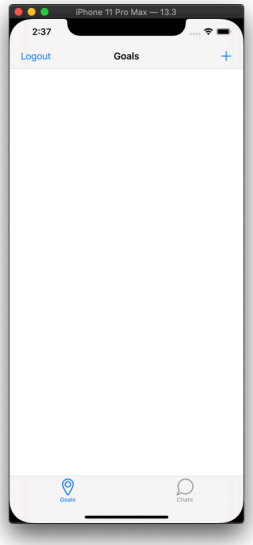
Adding the “+” button

Currently, there is no actual way to make a goal. The first thing I want to do is add a button to the navigation bar on the goal tracker screen.

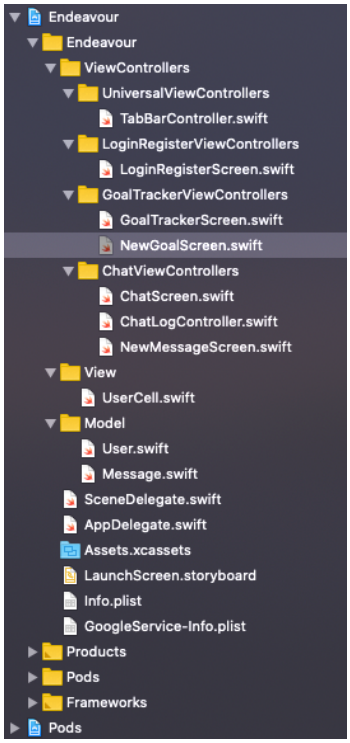
In “GoalTrackerSwift”, I’ve added the following code to “viewDidLoad()”:

```
26        //Add a "+" button to the top right of the navigation bar
27        navigationItem.rightBarButtonItem = UIBarButtonItem(barButtonSystemItem: .add, target: self, action: nil)
```

And the result:



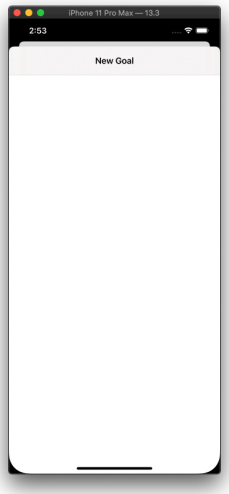
I now have a button I can work with.
The next thing I’m going to do is create a new class called “NewGoalScreen” which will be the interface the user interacts with when they want to create a goal.



Currently, the "+" button doesn't actually do anything. I need to write a function that will display an instance of "NewMessageScreen" when called and link it to the button.

```
33      @objc func displayNewGoalScreen() {
34          let newGoalScreen = NewGoalScreen()
35          let navigationControllerNewGoalScreen = UINavigationController(rootViewController: newGoalScreen)
36          present(navigationControllerNewGoalScreen, animated: true, completion: nil)
37      }
```

I've then passed #selector(displayNewGoalScreen) to the "+" button. Now, when I try to add a goal:



Note: I've set the title of "NewGoalScreen" to "New Goal" and I've set the background colour to white.

Building the “New Goal” interface

First, let’s add a cancel button:

```swift
import UIKit

class NewGoalScreen: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        //Set the background colour
        view.backgroundColor = .white

        //Set the view title
        title = "New Goal"

        //Adds a cancel button to the navigation bar
        navigationItem.leftBarButtonItem = UIBarButtonItem(barButtonSystemItem: .cancel, target: self, action: #selector(handleCancel))

    }


    //Dismisses the view when called
    @objc func handleCancel() {
        self.dismiss(animated: true, completion: nil)
    }

}
```

This interface is going to need four key elements:

1. A text field to input the goal’s name
2. A text field to input the email of the account keeping them accountable
3. A data/time picker for the goal’s deadline
4. A button to submit the goal

Let’s define these elements:

```swift
    /* Define all the UI elements */

    //Set up the "Goal Name" text field
    let goalNameTextField: UITextField = {
        let tf = UITextField()
        tf.placeholder = "Goal Name"
        tf.translatesAutoresizingMaskIntoConstraints = false
        return tf
    }()


    //Set up the "Accountablility" text field
    let accountabilityTextField: UITextField = {
        let tf = UITextField()
        tf.placeholder = "Peer's Email"
        tf.translatesAutoresizingMaskIntoConstraints = false
        return tf
    }()


    //Set up the UIDatePicker
    let goalEndDatePicker: UIDatePicker = {
        let dp = UIDatePicker()
        dp.translatesAutoresizingMaskIntoConstraints = false
        return dp
    }()


    //Set up the submit button
    let submitButton: UIButton = {
        let bn = UIButton()
        bn.translatesAutoresizingMaskIntoConstraints = false
        return bn
    }()
```

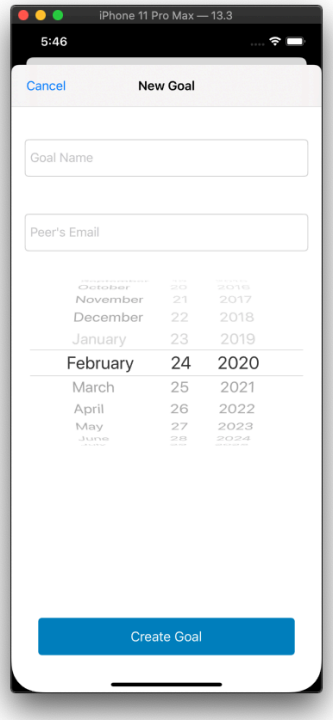
Now that they have been defined, I’ll add them one by one to the view and give them constraints.

```
    /* Defines the functions that will set up the UI constraints*/

    //Sets up the constraints for goalNameTextField
    func setupGoalNameTextField() {
        goalNameTextField.topAnchor.constraint(equalTo: view.topAnchor, constant: 100).isActive = true
        goalNameTextField.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
        goalNameTextField.heightAnchor.constraint(equalToConstant: 50).isActive = true
        goalNameTextField.widthAnchor.constraint(equalTo: view.widthAnchor, constant: -36).isActive = true
    }


    //Sets up th constraints for accountabilityTextField
    func setupAccountablilityTextField() {
        accountabilityTextField.topAnchor.constraint(equalTo: view.topAnchor, constant: 200).isActive = true
        accountabilityTextField.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
        accountabilityTextField.heightAnchor.constraint(equalToConstant: 50).isActive = true
        accountabilityTextField.widthAnchor.constraint(equalTo: view.widthAnchor, constant: -36).isActive = true
    }


    //Sets up the constraints for goalEndDatePicker
    func setupGoalEndDatePicker() {
        goalEndDatePicker.topAnchor.constraint(equalTo: view.topAnchor, constant: 250).isActive = true
        goalEndDatePicker.heightAnchor.constraint(equalToConstant: 300).isActive = true
        goalEndDatePicker.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
        goalEndDatePicker.widthAnchor.constraint(equalTo: view.widthAnchor, constant: -50).isActive = true
    }


    //Sets up the constraints for the submit button
    func setupSubmitButton() {
        submitButton.bottomAnchor.constraint(equalTo: view.bottomAnchor, constant: -50).isActive = true
        submitButton.widthAnchor.constraint(equalTo: view.widthAnchor, constant: -72).isActive = true
        submitButton.heightAnchor.constraint(equalToConstant: 50).isActive = true
        submitButton.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
    }
```

I'll then add them as sub-views to the "viewDidLoad" method and call the functions shown above.

```
override func viewDidLoad() {
    super.viewDidLoad()

    //Set the background colour
    view.backgroundColor = .white

    //Set the view title
    title = "New Goal"

    //Adds a cancel button to the navigation bar
    navigationItem.leftBarButtonItem = UIBarButtonItem(barButtonSystemItem: .cancel, target: self, action: #selector(handleCancel))

    //Adds all the subviews
    view.addSubview(goalNameTextField)
    view.addSubview(accountabilityTextField)
    view.addSubview(goalEndDatePicker)
    view.addSubview(submitButton)

    //Set up all UI constraints
    setupGoalNameTextField()
    setupAccountablilityTextField()
    setupGoalEndDatePicker()
    setupSubmitButton()
}
```

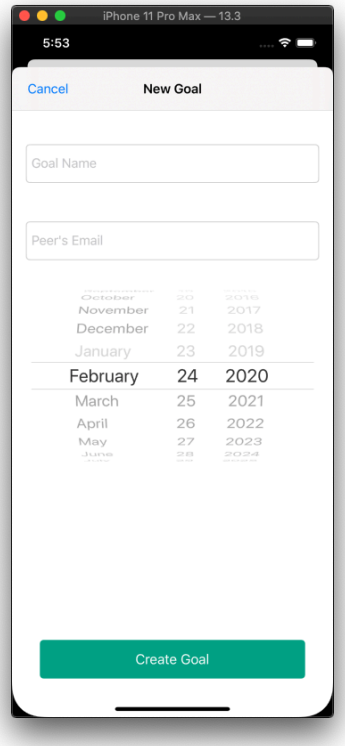
After some tinkering and tweaking, to the set-up functions, this is what I end up with:



The modified set-up functions are shown below:

```swift
/* Define all the UI elements */

//Set up the "Goal Name" text field
let goalNameTextField: UITextField = {
    let tf = UITextField()
    tf.placeholder = "Goal Name"
    tf.borderStyle = .roundedRect
    tf.translatesAutoresizingMaskIntoConstraints = false
    return tf
}()


//Set up the "Accountablility" text field
let accountabilityTextField: UITextField = {
    let tf = UITextField()
    tf.placeholder = "Peer's Email"
    tf.borderStyle = .roundedRect
    tf.translatesAutoresizingMaskIntoConstraints = false
    return tf
}()


//Set up the UIDatePicker
let goalEndDatePicker: UIDatePicker = {
    let dp = UIDatePicker()
    dp.datePickerMode = UIDatePicker.Mode.date
    dp.minimumDate = NSDate() as Date
    dp.translatesAutoresizingMaskIntoConstraints = false
    return dp
}()


//Set up the submit button
let submitButton: UIButton = {
    let bn = UIButton()
    bn.backgroundColor = UIColor(displayP3Red: 41/255, green: 128/255, blue: 185/255, alpha: 1)
    bn.setTitleColor(UIColor.white, for: .normal)
    bn.setTitle("Create Goal", for: .normal)
    bn.layer.cornerRadius = 5
    bn.translatesAutoresizingMaskIntoConstraints = false
    return bn
}()
```

I've added borders to both input text fields, set a minimum date (so people can't set their goals' due dates to anypoint before the day in which their goals where made) and the date style for the date picker, and I've given the button some colour, a title, and rounded corners. I think I might change the button colour to green:



The UI for creating goals is now complete; onto actually creating goals!

Creating a goal

I need to write a function which takes the input from the above UI when the user presses the "Create Goal" button and uploads it to Firebase.

```swift
    //Uploads the goal to Firebase and creates a reference for it under "User-Goals"
    @objc func createNewGoal() {

        //Creates a reference to a node called "Goals" in Firebase
        let ref = Database.database().reference().child("Goals")

        //Sets up another reference, this time to an auto-generate child node of "Goals"
        let childRef = ref.childByAutoId()

        //Get the values found in the various input UI elements
        let goalName = goalNameTextField.text!
        let email = accountabilityTextField.text!
        let startDate = NSDate()
        let endDate = goalEndDatePicker.date

        //Store the user input as a dictionary
        let values = ["name": goalName, "accountabilityEmail": email, "startDate": startDate, "endDate": endDate] as [String : Any]

        //Upload the user input to Firebase as a goal
        childRef.updateChildValues(values) { (error, ref) in

            //Checks for and catches any errors
            if error != nil{
                print(error as Any)
                return
            }

            //Unwraps goalID
            guard let goalID = childRef.key else {
                return
            }

            //Gets the current users unique Firebase ID
            let currentUID = Auth.auth().currentUser!.uid

            //Creates a new database reference to "currentUID" as a sub-node of "User Goals"
            let userGoalRef = Database.database().reference().child("User-Goals").child(currentUID)

            //Uploads the goal ID as a sub-node of the user's ID
            userGoalRef.updateChildValues([goalID: true])
        }

        //Dismisses itself.
        handleCancel()
    }
```

This function uses a technique called “fan-out” to overcome the issues of not having a relational database. Because the database is flat file, a you want everything to be as flat as possible. You want to avoid nesting lists within nodes. This technique solves this problem by creating reference nodes that connect two different entities together. The function above creates a node, “Goals”, where all the goals created are stored. It then creates another node, “User-Goals”, which it uses to link each user to the goals they have created, simulating the relationship between two entities in a relational database, where the goal ID serves as the foreign key of the reference node.

When I run the program and try to add a goal, I get this error:

```
2020-02-25 17:47:23.445120+0000 Endeavour[4737:509907] *** Terminating app due to uncaught exception 'InvalidFirebaseData', reason:
    '(updateChildValues:withCompletionBlock:) Cannot store object of type __NSTaggedDate at . Can only store objects of type NSNumber,
    NSString, NSDictionary, and NSArray.'
```

The issue seems to lie in storing an variable of type NSDate in firebase. I’ll try converting the NSDate type to a string and storing that instead.

Turns out, that was a much more complex process than I anticipated.

I tried converting the dates to strings using an instance of “DateFormatter()” but it still didn’t seem to work. After running through the “createNewGoal” function several times using a breakpoint and stepping through the code line by line, I discovered that the strings I had converted the dates into didn’t have any values for some strange reason. For the start date, I think it was because I was trying to convert an “NSDate” object to a “Date” object

before then trying to convert it to a string which was getting messy. For the “endDate”, I didn’t seem to be actually getting anything from the “goalEndDatePicker” object.

After hunting around on Stack Overflow and looking through Apple’s Developer Documentation, I came up with the following solution.

First, I added a target to “goalEndDatePicker”:

```
38      //Set up the UIDatePicker
39      let goalEndDatePicker: UIDatePicker = {
40          let dp = UIDatePicker()
41          dp.datePickerMode = UIDatePicker.Mode.date
42          dp.minimumDate = NSDate() as Date
43          dp.addTarget(self, action: #selector(changeGoalPickerDate(datePicker:)), for: .valueChanged)
44          dp.translatesAutoresizingMaskIntoConstraints = false
45          return dp
46      }()
```

So now whenever the date on the “UIDatePicker” instance is changed, it runs the “changeGoalPickerDate” function.

```
96      //Sets the end date of the goal based on the goal picker reading
97      @objc func changeGoalPickerDate(datePicker: UIDatePicker) {
98          let dateFormatter = DateFormatter()
99          dateFormatter.dateFormat = "dd/MM/yyyy"
100         goalPickerDate = dateFormatter.string(from: datePicker.date)
101     }
```

This function changes the value of a variable called “goalPickerDate” to whatever the date displayed on “goalEndDatePicker” is.

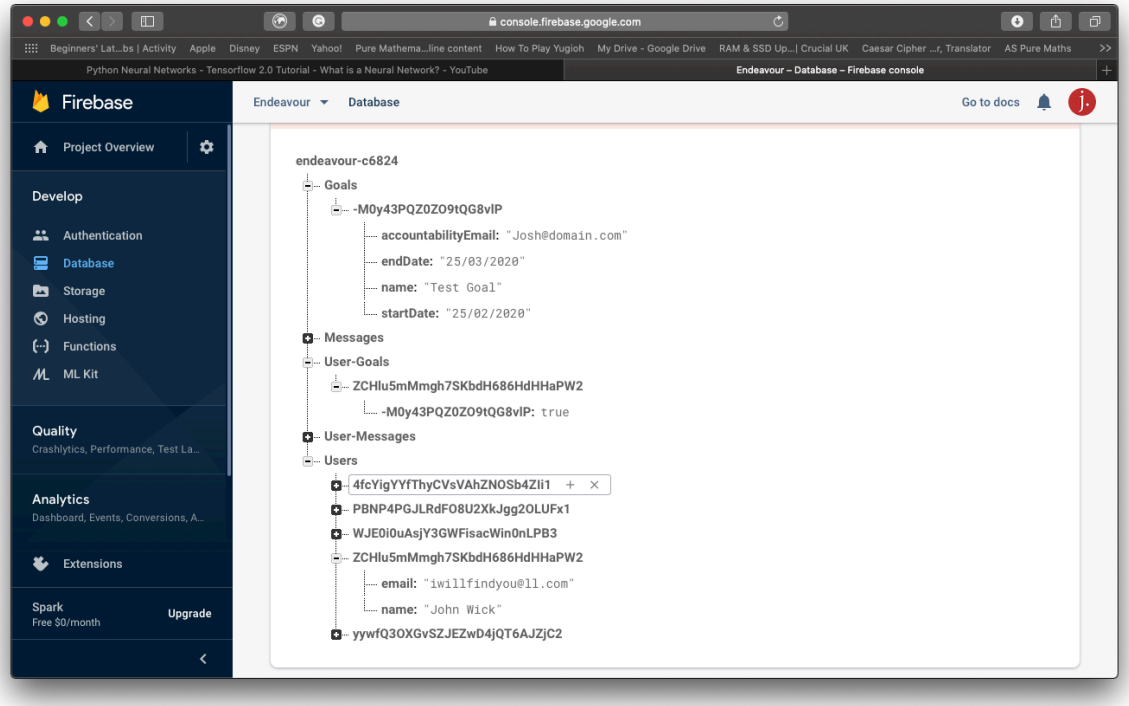
Putting that to one side, I wrote another function, “getCurrentDate”, which returns the current date as a string:

```
104     //Gets the current date and returns it as a string
105     func getCurrentDate() -> String {
106         let currentDate = Date()
107         let dateFormatter = DateFormatter()
108         dateFormatter.dateFormat = "dd/MM/yyyy"
109         let currentDateString = dateFormatter.string(from: currentDate)
110         return currentDateString
111     }
```

I then returned to the “createNewGoal” function and set the values of both “startDate” and “endDate”:

```swift
    //Uploads the goal to Firebase and creates a reference for it under "User-Goals"
    @objc func createNewGoal() {

        //Creates a reference to a node called "Goals" in Firebase
        let ref = Database.database().reference().child("Goals")

        //Sets up another reference, this time to an auto-generate child node of "Goals"
        let childRef = ref.childByAutoId()

        //Get the values found in the various input UI elements
        let goalName = goalNameTextField.text!
        let email = accountabilityTextField.text!
        let startDate = getCurrentDate()
        let endDate = goalPickerDate

        //Store the user input as a dictionary
        let values = ["name": goalName, "accountabilityEmail": email, "startDate": startDate, "endDate": endDate] as [String : Any]

        //Upload the user input to Firebase as a goal
        childRef.updateChildValues(values) { (error, ref) in

            //Checks for and catches any errors
            if error != nil{
                print(error as Any)
                return
            }

            //Unwraps goalID
            guard let goalID = childRef.key else {
                return
            }

            //Gets the current users unique Firebase ID
            let currentUID = Auth.auth().currentUser!.uid

            //Creates a new database reference to "currentUID" as a sub-node of "User Goals"
            let userGoalRef = Database.database().reference().child("User-Goals").child(currentUID)

            //Uploads the goal ID as a sub-node of the user's ID
            userGoalRef.updateChildValues([goalID: true])
        }

        //Dismisses itself.
        handleCancel()
    }
```

Now when I run the code and add a date:



Firebase updates! (I am currently logged in as John Wick, which is why the goal is linked to his user ID).

Retrieving goals

To retrieve the goals from Firebase, I can use a very similar method to the one I used to fetch messages.

The first thing I've done is make "GoalTrackerScreen" inherit from "UITableViewController" so that I can display each goal as a cell in the table view.

```
12  class GoalTrackerScreen: UITableViewController {
```

I've then created two further attributes for "GoalTrackerScreen":

```
14      let cellID = "cellID"
15
16      var goals = [Goal]()
```

"cellID" will be the default ID for any table cell created. All the user's goals pulled down from Firebase will be placed inside of the list "goals" as "goal" objects.

After doing that, I've modified the "tableView" functions to cater to the goals stored in "Goals":

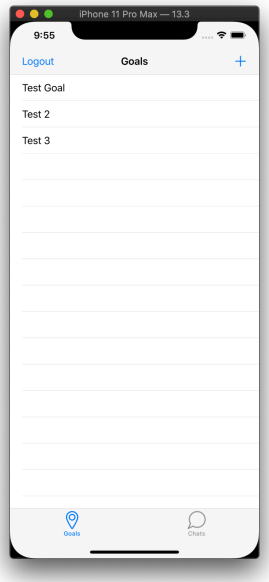
```
177     //Sets the information of each cell to match each goal
178     override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
179         let cell = UITableViewCell()
180         let goal = goals[indexPath.row]
181
182         cell.textLabel?.text = goal.name
183         cell.detailTextLabel?.text = goal.endDate
184
185         return cell
186     }
```

This is quite rudimentary at the moment, but I will improve upon the display once I have successfully retrieved the user's goals from Firebase.

The Next pages contains all the code for the "observeUserGoals()" function.

```swift
//Retrieves all the user's current goals from Firebase
func observeUserGoals() {

    //Checks the current user is logged in
    guard let uid = Auth.auth().currentUser?.uid else {
        return
    }

    //Sets up a reference to the user's personal node that extends "User-Goals"
    let ref = Database.database().reference().child("User-Goals").child(uid)

    //Observes all the goal IDs stored at that node.
    ref.observe(.childAdded, with: { (snapshot) in

        //Sets "goalID" equal to the unique ID of each goal stored at "User-Goals"->uid
        let goalID = snapshot.key

        //Sets up another database reference to each goal's ID but within "Goals"
        let goalReference = Database.database().reference().child("Goals").child(goalID)

        //Retrieves the data for each goal, stores it as an instance of "Goal", and then adds it
        //to "goals[]"
        goalReference.observe(.value, with: { (snapshot) in

            if let dictionary = snapshot.value as? [String: AnyObject] {

                let goal = Goal()

                goal.name = dictionary["name"] as? String
                goal.startDate = dictionary["startDate"] as? String
                goal.endDate = dictionary["endDate"] as? String
                goal.accountabilityEmail = dictionary["email"] as? String

                self.goals.append(goal)
            }

            //Refreshes the table view
            self.tableView.reloadData()

        }, withCancel: nil)

    }, withCancel: nil)

}
```

Running the simulator:



Now that the goals are being displayed, I'm just going to improve their aesthetic a little.

In the "View" folder, I've created a new file, "Goal.swift". Within "Goal.swift", I've created a new class:

```
import UIKit
import Firebase

class GoalCell: UITableViewCell {

    //Give GoalCell the attribute "goal" of type "Goal?"
    var goal: Goal? {

        //"didSet" will run as soon as "goal" is assigned a value
        didSet {

            //Sets the text of the cell to the goal's name
            self.textLabel?.text = goal?.name

            //Unwraps the endDate property of goal
            guard let endDate = goal?.endDate else{
                return
            }

            //Sets the subtext of the cell to the end date and alters its colour
            self.detailTextLabel?.text = "End date: " + endDate
            self.detailTextLabel?.textColor = UIColor.darkGray
        }
    }


    //Changes the default style of the cell.
    override init(style: UITableViewCell.CellStyle, reuseIdentifier: String?) {
        super.init(style: .subtitle, reuseIdentifier: reuseIdentifier)

    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

}
```

"GoalCell" inherits from "UITableViewCell", allowing me to register it as a custom cell.

Within the "viewDidLoad" method of "GoalTrackerScreen", I've registered "GoalCell" as the default "UITableViewCell" class.

```
        //Register custom class "GoalCell" as the default cell class
        tableView.register(GoalCell.self, forCellReuseIdentifier: cellID)
```

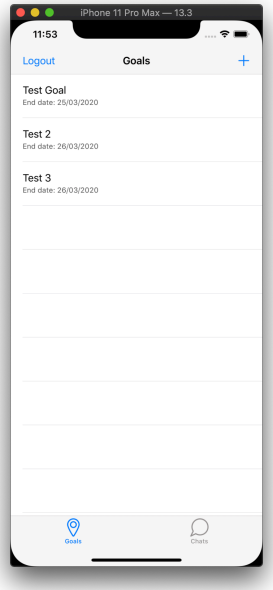
I've then modified one of the tableView override function to work around the "GoalCell" class:

```swift
    //Sets the information of each cell to match each goal
    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
        UITableViewCell {

        //Creates an instance of the GoalCell class for each value of indexPath
        let cell = tableView.dequeueReusableCell(withIdentifier: cellID, for: indexPath) as! GoalCell

        //Gets each goal from the goals list where the index of the item fetched
        //matches the row number in the table
        let goal = goals[indexPath.row]
        //Sets the value of the "goal" attribute in "GoalCell" to the "goal" constant
        cell.goal = goal

        return cell
    }
```

I've also just bumped up the height of each cell:

```swift
    //Make the cells in the table bigger
    override func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat
        {
        return 72
    }
```

The simulator now gives me this:



Completing goals

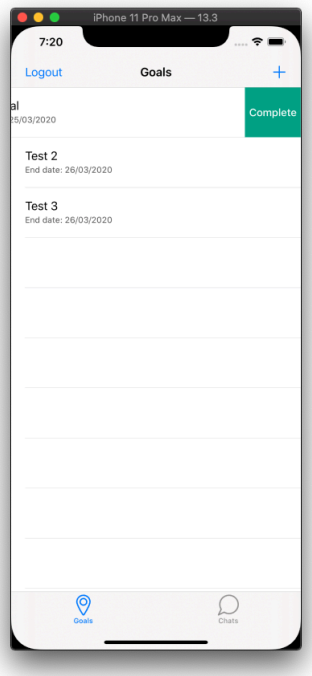
One of the most important factors about setting goals is having the ability to complete them. Ideally, a user would be able to complete a goal by swiping it off the screen. I've added the following code to the end of "GoalTrackerScreen.swift":

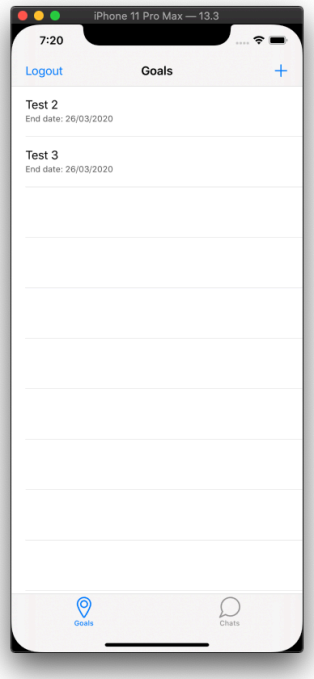
```swift
//Allows the user to edit the table
override func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool {
    return true
}

//Configures the action taken when a use swipes a cell.
override func tableView(_ tableView: UITableView, trailingSwipeActionsConfigurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration? {

    //Creates a new contextual action and set's its basic attributes
    let deleteAction = UIContextualAction(style: .destructive, title: "Complete") { (_, _, complete) in

        //When the action is called, the "goalCell" swiped is removed from the "goals" list and from the table
        self.goals.remove(at: indexPath.row)
        self.tableView.deleteRows(at: [indexPath], with: .automatic)

        complete(true)
    }

    //Changes the background colour of the delete button
    deleteAction.backgroundColor = UIColor(displayP3Red: 22/255, green: 160/255, blue: 133/255, alpha: 1.0)

    //Sets "deleteAction" as the action taken when a cell is swiped completely to the left or right.
    let configuration = UISwipeActionsConfiguration(actions: [deleteAction])
    configuration.performsFirstActionWithFullSwipe = true

    return configuration
}
```

The first “override” statement (line 161) alters the tableView so the user has permission to edit the table.

The second override function (line 166) used to modify the action when a cell is swiped, creating a new “completion swipe” that allows the user to remove a completed goal from both the “goals” list and from the “tableView”.

The simulator now looks like this (when I swipe a goal):





This only superficially “completes” a goal as it only removes it from the view and the “goals” list; the goal is still stored in Firebase and so will simply reappear when the user refreshes the “GoalTrackerScreen”. When the user swipes a cell, I need it to call a function that also removes the goal from Firebase.

I’ve added “goalID” as an attribute of the “Goal” class:

```swift
import UIKit

class Goal: NSObject {

    var goalID: String?
    var name: String?
    var accountabilityEmail: String?
    var startDate: String?
    var endDate: String?

}
```

In “GoalTrackingScreen.swift” -> “GoalTrackerScreen” -> “observeUserGoals()”, I’ve added line 115:

```swift
func observeUserGoals() {

    //Checks the current user is logged in
    guard let uid = Auth.auth().currentUser?.uid else {
        return
    }

    //Sets up a reference to the user's personal node that extends "User-Goals"
    let ref = Database.database().reference().child("User-Goals").child(uid)

    //Observes all the goal IDs stored at that node.
    ref.observe(.childAdded, with: { (snapshot) in

        //Sets "goalID" equal to the unique ID of each goal stored at "User-Goals"->uid
        let goalID = snapshot.key

        //Sets up another database reference to each goal's ID but within "Goals"
        let goalReference = Database.database().reference().child("Goals").child(goalID)

        //Retrieves the data for each goal, stores it as an instance of "Goal", and then adds it
        //to "goals[]"
        goalReference.observe(.value, with: { (snapshot) in

            if let dictionary = snapshot.value as? [String: AnyObject] {

                let goal = Goal()

                goal.name = dictionary["name"] as? String
                goal.startDate = dictionary["startDate"] as? String
                goal.endDate = dictionary["endDate"] as? String
                goal.accountabilityEmail = dictionary["email"] as? String
                goal.goalID = goalID
```

I can now use this attribute to reference the goal in Firebase so that I can remove it once it has been completed.

Now, onto the actual removal function.

In “GoalTrackerScreen”, I’ve created a function called, “removeGoal()” that takes a goalID as an argument.

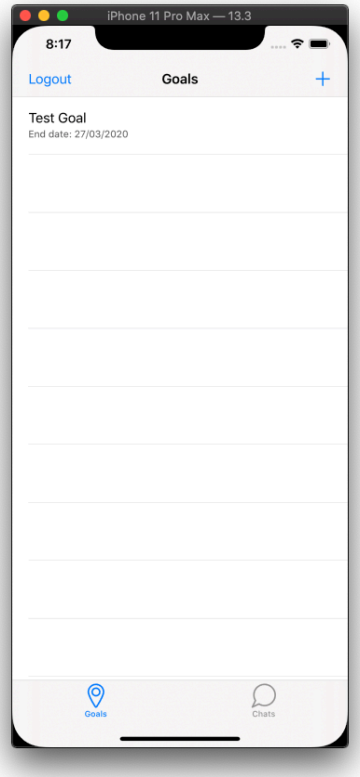
```swift
    //Removes a goal from Firebase
    func removeGoal(withID goalID: String) {

        //Unwrap the optional userID
        guard let userID = Auth.auth().currentUser?.uid else {
            return
        }

        //Create references to Firebase, one to the goal in the "Goals" sub-tree and the other to the
        //goal in the "User-Goals" sub-tree
        let goalRef = Database.database().reference().child("Goals").child(goalID)
        let userGoalRef = Database.database().reference().child("User-Goals").child(userID).child(goalID)

        //Removes the value specified in the "goalRef" reference
        goalRef.removeValue { (error, _) in

            //Catches and prints any errors
            if error != nil {
                print(error?.localizedDescription as Any)
            }

        }

        //Removes the value specified in the "userGoalRef" reference
        userGoalRef.removeValue { (error, _) in

            //Catches and prints any errors
            if error != nil {
                print(error?.localizedDescription as Any)
            }
        }
    }
```

It then uses the “goalID” to locate that specific goal in all its instances in Firebase and then removes them. I have then called “removeGoal” in the override function responsible for the removal of the “goalCell” from the view.

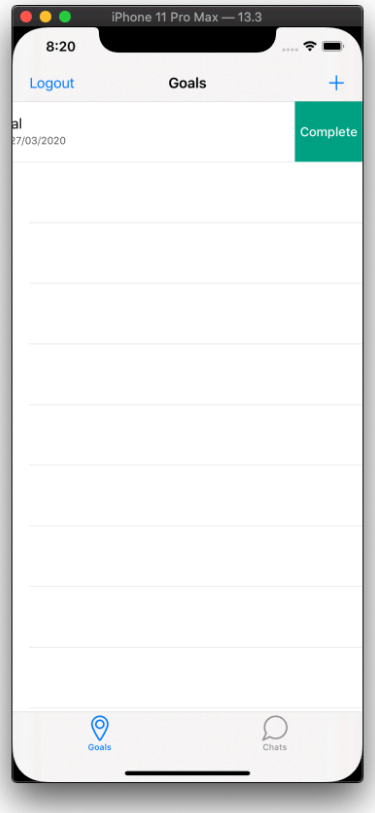
```swift
    //Configures the action taken when a use swipes a cell.
    override func tableView(_ tableView: UITableView, trailingSwipeActionsConfigurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration? {

        //Creates a new contextual action and set's its basic attributes
        let deleteAction = UIContextualAction(style: .destructive, title: "Complete") { (_, _, complete) in

            //Retrieves the "goalID" of the goal to be removed and unwraps it.
            guard let goalID = self.goals[indexPath.row].goalID else {
                return
            }

            //When the action is called, the "goalCell" swiped is removed from the "goals" list, from the table, and from Firebase.
            self.goals.remove(at: indexPath.row)
            self.tableView.deleteRows(at: [indexPath], with: .automatic)
            self.removeGoal(withID: goalID)

            complete(true)
        }
```

Now, to test it.

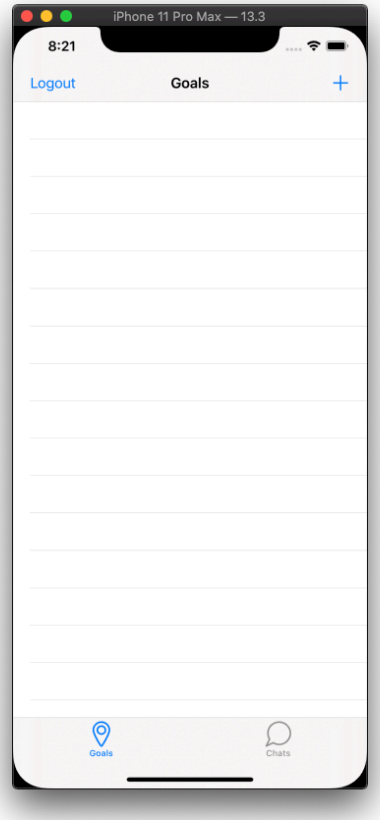
Note: I’ve deleted all the goals and “User-Goal” references from Firebase just to make sure the early goals that were implemented before the “User-Goal” referencing system came into effect didn’t clutter up the simulator.

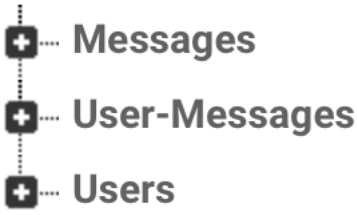


- Goals
  - -M17I2g8qf3X0xzLOxcS
    - accountabilityEmail: "Email"
    - endDate: "27/03/2020"
    - name: "Test Goal"
    - startDate: "27/02/2020"
  - Messages
  - User-Goals
    - ZCHlu5mMmgh7SKbdH686HdHHaPW2
      - -M17I2g8qf3X0xzLOxcS: true
  - User-Messages
  - Users



- Goals
  - -M17I2g8qf3X0xzLOxcS
    - accountabilityEmail: "Email"
    - endDate: "27/03/2020"
    - name: "Test Goal"
    - startDate: "27/02/2020"
  - Messages
  - User-Goals
    - ZCHlu5mMmgh7SKbdH686HdHHaPW2
      - -M17I2g8qf3X0xzLOxcS: true
  - User-Messages
  - Users





And it works just fine.

Missing a goal's deadline

If the user misses a goal's deadline, their "peer" (the person holding them accountable) needs to be notified.

```swift
//This function will iterate throught the "goals" list and compare each goal's end date to today's
func checkGoalEndDate(goal: Goal) {

    let currentDate = NSDate() as Date
    let dateFormatter = DateFormatter()
    dateFormatter.dateFormat = "dd/MM/yyyy"

    guard let endDate = dateFormatter.date(from: goal.endDate!) else {
        return
    }

    //Compare dates
    if currentDate > endDate {

        //Set up a reference to "Users" in Firebase
        let ref = Database.database().reference().child("Users")

        //Observe all the values stored at the "Users" node
        ref.observe(.value, with: { (snapshot) in

            //Unwrapps the snapshot as a Dictionary
            if let dictionary = snapshot.value as? [String: AnyObject] {

                //Itterates through the dictionary
                for item in dictionary {

                    //Unwrapps the email attribute of the "goal" argument
                    guard let goalEmail = goal.accountabilityEmail else {
                        return
                    }

                    //Unwraps the email retrieved from Firebase
                    guard let retrievedEmail = item.value["email"] as? String? else {
                        return
                    }

                    //Checks to see if the email from Firebase matches the email stored as an attribute of "goal"
                    if goalEmail == retrievedEmail {

                        let message = "Hey, I've missed a deadline! Chase me up!"
                        let peerID = item.key

                        //Sends a message to the goal's linked user if the goal has passed it's deadline.
                        self.sendMessage(messageText: message, recipientID: peerID)

                    }

                }

            }

        }, withCancel: nil)

    }

}
```

The above function will do just that.

“checkGoalEndDate” is within “GoalTrackerScreen.swift” It takes a goal as input, checks the “endDate” attribute of that goal and compares it to the current date. If the goal’s deadline has been and gone, then the “email” attribute is compared to the user emails in Firebase. If a match is found, a text is sent to that person, notifying them of their friend’s missed deadline.

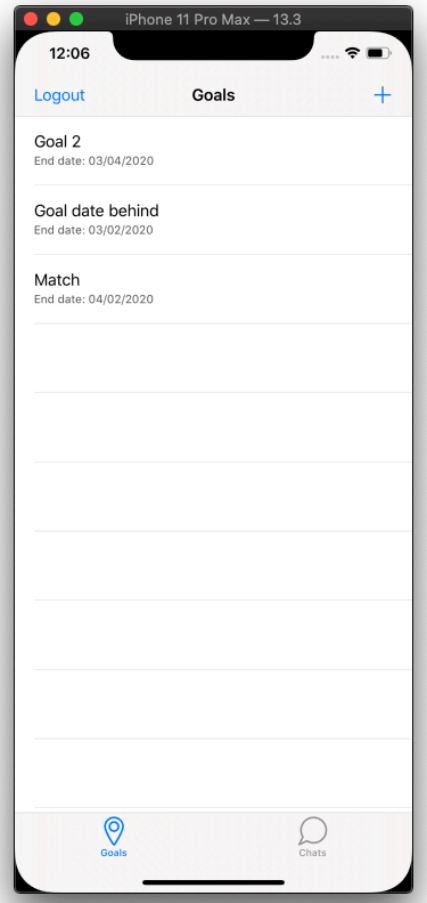
I’ve called this function within “observeUserGoals()” in “GoalTrackerScreen.swift”.

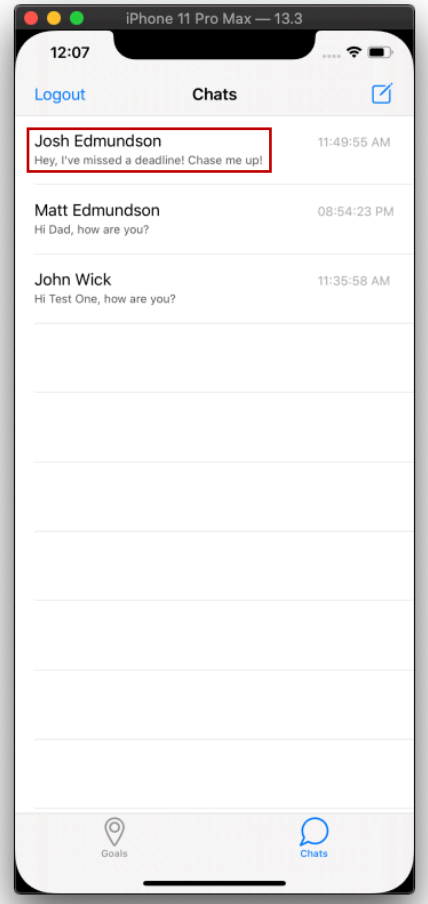
```
83      //Retrieves all the user's current goals from Firebase
84      func observeUserGoals() {
85
86          //Checks the current user is logged in
87          guard let uid = Auth.auth().currentUser?.uid else {
88              return
89          }
90
91          //Sets up a reference to the user's personal node that extends "User-Goals"
92          let ref = Database.database().reference().child("User-Goals").child(uid)
93
94          //Observes all the goal IDs stored at that node.
95          ref.observe(.childAdded, with: { (snapshot) in
96
97              //Sets "goalID" equal to the unique ID of each goal stored at "User-Goals"->uid
98              let goalID = snapshot.key
99
100             //Sets up another database reference to each goal's ID but within "Goals"
101             let goalReference = Database.database().reference().child("Goals").child(goalID)
102
103             //Retrieves the data for each goal, stores it as an instance of "Goal", and then adds it
104             //to "goals[]"
105             goalReference.observe(.value, with: { (snapshot) in
106
107                 if let dictionary = snapshot.value as? [String: AnyObject] {
108
109                     let goal = Goal()
110
111                     goal.name = dictionary["name"] as? String
112                     goal.startDate = dictionary["startDate"] as? String
113                     goal.endDate = dictionary["endDate"] as? String
114                     goal.accountabilityEmail = dictionary["accountabilityEmail"] as? String
115                     goal.goalID = goalID
116
117                     self.checkGoalEndDate(goal: goal)
118                     self.goals.append(goal)
119                 }
120
121                 //Refreshes the table view
122                 self.tableView.reloadData()
123
124             }, withCancel: nil)
125
126         }, withCancel: nil)
127
128     }
```

I've highlighted the line I've added.

The reason I called the function here was to cut down on processing time. The function "observeUserGoals()" has already referenced Firebase and created a goal object for each of the current user's goals. Calling "checkGoalEndDate" here saves me having to re-reference these goals later on.

Now, when I build and run the simulator:



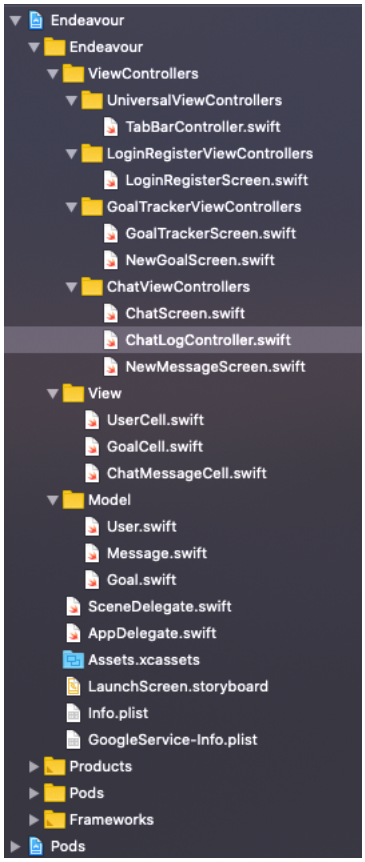


John Wick has failed to meet one of his goals, a goal which he linked to Josh Edmundson's account, and so the app has sent a message to Josh Edmundson to let him know.

Note, though all three of John Wick's goal end dates have passed, he only sent one message. This is because the two other goals were either not linked to an email at all, or the email wasn't in the database.

Complete code

Below I've copied and pasted all the code I've written for this project.



TabBarController.swift:

```
import UIKit

class TabBarController: UITabBarController {

    override func viewDidLoad() {
        super.viewDidLoad()

        //Sets up the tab bar
        setupTabBar()
    }


    //Creates two navigation controllers and places them whithin our
custom view controllers
    func setupTabBar() {

        //Creates two instances of UINavigationController within the
screens
        let goalTrackerScreen =
UINavigationController(rootViewController: GoalTrackerScreen())
        let chatScreen = UINavigationController(rootViewController:
ChatScreen())
```

```
        //Creates a tabBarItem for each of the two screens. Add
icons later.
        goalTrackerScreen.tabBarItem.title = "Goals"
        goalTrackerScreen.tabBarItem.image = UIImage(named:
"goals_pin_unselected")
        goalTrackerScreen.tabBarItem.selectedImage = UIImage(named:
"goals_pin_selected")

        chatScreen.tabBarItem.title = "Chats"
        chatScreen.tabBarItem.image = UIImage(named:
"messages_unselected")
        chatScreen.tabBarItem.selectedImage = UIImage(named:
"messages_selected")

        //Adds the view controllers to the tab bar
        viewControllers = [goalTrackerScreen, chatScreen]
    }



}
```

LoginRegisterScreen.swift:

```
import UIKit
import Firebase

class LoginRegisterScreen: UIViewController {

    var chatScreen: ChatScreen?

    //Creates the container to hold the text fields
    let inputsContainerView: UIView = {
        let view = UIView()
        view.backgroundColor = UIColor.white
        view.translatesAutoresizingMaskIntoConstraints = false
        view.layer.cornerRadius = 5
        view.layer.masksToBounds = true
        return view
    }()


    //Create the input field for the user's name.
    let nameTextField: UITextField = {
        let tf = UITextField()
        tf.placeholder = "Name"
        tf.translatesAutoresizingMaskIntoConstraints = false
        return tf
    }()
```

```
    //This will create a UIView to seperate the name text field from
the emailTextField
    let nameSeparatorView: UIView = {
       let view = UIView()
        view.backgroundColor = UIColor(displayP3Red: 220/255, green:
220/255, blue: 220/255, alpha: 1)
        view.translatesAutoresizingMaskIntoConstraints = false
        return view
    }()


    //Create email text field
    let emailTextField: UITextField = {
        let tf = UITextField()
        tf.placeholder = "Email"
        tf.translatesAutoresizingMaskIntoConstraints = false
        return tf
    }()

    //Create the email and password text field seperator
    let emailSeparatorView: UIView = {
       let view = UIView()
        view.backgroundColor = UIColor(displayP3Red: 220/255, green:
220/255, blue: 220/255, alpha: 1)
        view.translatesAutoresizingMaskIntoConstraints = false
        return view
    }()


    //Creates the password text field
    let passwordTextField: UITextField = {
        let tf = UITextField()
        tf.placeholder = "Password"
        tf.translatesAutoresizingMaskIntoConstraints = false
        tf.isSecureTextEntry = true //This will hide the user input.
        return tf
    }()


    //Create the login/register button
    let loginRegisterButton: UIButton = {
        let button = UIButton(type: .system)
        button.backgroundColor = UIColor(displayP3Red: 41/255,
green: 128/255, blue: 185/255, alpha: 1)
        button.setTitle("Register", for: .normal)
        button.setTitleColor(UIColor.white, for: .normal)
        //Always set this to false
        button.translatesAutoresizingMaskIntoConstraints = false
        button.titleLabel?.font = UIFont.boldSystemFont(ofSize: 16)

        //Makes the button run the handleRegisterFunction when the
user releases the button within its boundary
```

```
        button.addTarget(self, action:
#selector(handleLoginRegister), for: .touchUpInside)

        return button
    }()


    //Create segmented controller
    let loginRegisterSegmentedControl: UISegmentedControl = {
        let sc = UISegmentedControl(items: ["Login", "Register"])
        sc.translatesAutoresizingMaskIntoConstraints = false
        sc.tintColor = UIColor.white
        sc.selectedSegmentIndex = 1
        sc.addTarget(self, action:
#selector(handleLoginRegisterChange), for: .valueChanged)
        return sc
    }()


    override func viewDidLoad() {
        super.viewDidLoad()

        //Change the background colour to "Green Sea"
        view.backgroundColor = UIColor(displayP3Red: 22/255, green:
160/255, blue: 133/255, alpha: 1.0)

        //Add all subviews to the screen
        view.addSubview(inputsContainerView)
        view.addSubview(loginRegisterButton)
        view.addSubview(loginRegisterSegmentedControl)


        //Create subview dimensions using constraints
        setupInputsContainerView()
        setupLoginRegisterButton()
        setupLoginRegisterSegmentedControl()
    }


    //This is called when switiching between segments in the
segmented control.
    @objc func handleLoginRegisterChange() {
        //Set up two variables base on the current segment selected
        let currentIndex =
loginRegisterSegmentedControl.selectedSegmentIndex
        let title =
loginRegisterSegmentedControl.titleForSegment(at: currentIndex)

        //Update the text on the loginRegisterButton accordingly
        loginRegisterButton.setTitle(title, for: .normal)

        //Change height of inputContainerView based on the current
index
```

```
        inputsContainerViewHeightAnchor?.constant = currentIndex ==
0 ? 100 : 150

        //Change height of nameTextField
        nameTextFieldHeightAnchor?.isActive = false
        nameTextFieldHeightAnchor =
nameTextField.heightAnchor.constraint(equalTo:
inputsContainerView.heightAnchor, multiplier: currentIndex == 0 ? 0
: 1/3)
        nameTextFieldHeightAnchor?.isActive = true

        //Change the height of the emailTextField
        emailTextFieldHeightAnchor?.isActive = false
        emailTextFieldHeightAnchor =
emailTextField.heightAnchor.constraint(equalTo:
inputsContainerView.heightAnchor, multiplier: currentIndex == 0 ?
1/2 : 1/3)
        emailTextFieldHeightAnchor?.isActive = true

        //Change the height of the passwordTextField
        passwordTextFieldHeightAnchor?.isActive = false
        passwordTextFieldHeightAnchor =
passwordTextField.heightAnchor.constraint(equalTo:
inputsContainerView.heightAnchor, multiplier: currentIndex == 0 ?
1/2 : 1/3)
        passwordTextFieldHeightAnchor?.isActive = true
    }


    //This function determins whether the user is trying to login
    //or register and runs the corresponding function
    @objc func handleLoginRegister() {
        if loginRegisterSegmentedControl.selectedSegmentIndex == 0 {
            handleLogin()
        } else {
            handleRegister()
        }
    }


    //This function will sign the user in.
    func handleLogin() {
        guard let email = emailTextField.text, let password =
passwordTextField.text else {
            print("Form is not valid")
            return
        }

        Auth.auth().signIn(withEmail: email, password: password,
completion: {
            (user, error) in

            if error != nil {
                print(error as Any)
```

```
                return
            }

            //self.cleanUpChatScreenTable()

            self.dismiss(animated: true, completion: nil)

        })
    }


    //This function attempts to refresh the list of chats when you
log back in
    //but unfortunately, it doesn't work.
    func cleanUpChatScreenTable() {
        print("logging in")
        self.chatScreen?.messages.removeAll()
        self.chatScreen?.messagesDictionary.removeAll()
        self.chatScreen?.tableView.reloadData()
        self.chatScreen?.observeUserMessages()
    }


    //This function will register a new user
    @objc func handleRegister() {
        //This checks the inputs for email and password are actually
valid
        guard let email = emailTextField.text, let password =
passwordTextField.text, let name = nameTextField.text else {
            print("Form is not valid")
            return
        }

        //This creates a Firebase user using the inputs in the email
and password fields.
        Auth.auth().createUser(withEmail: email, password: password)
{ authResult, error in
            //This if statement is there to catch any errors
            if let err = error {
                print(err)
                return
            }

            //Save the new user to the database
            //Here, we create a variable that will reference the
database.
            var ref: DatabaseReference!
            ref = Database.database().reference(fromURL:
"https://endeavour-c6824.firebaseio.com/")

            //Create a dictionary of the values we want to store in
the database
            let values: Dictionary = ["name": name, "email": email]
            let userID = Auth.auth().currentUser!.uid
```

```
            //Add the new user's name and email to the database
            //to the database beneath the child nodes "Users"
            //and a unique id generated by childByAutoId

ref.child("Users").child(userID).updateChildValues(values,
withCompletionBlock: {
                (error, ref) in

                if error != nil {
                    print(error!)
                    return
                }

                print("Saved user sccessfully")
                self.dismiss(animated: true, completion: nil)
            })

        }

    }


    //Set up constraints for loginRegisterSegmentedControl
    func setupLoginRegisterSegmentedControl() {
        //Setup constraints

loginRegisterSegmentedControl.centerXAnchor.constraint(equalTo:
view.centerXAnchor).isActive = true

loginRegisterSegmentedControl.bottomAnchor.constraint(equalTo:
inputsContainerView.topAnchor, constant: -12).isActive = true

loginRegisterSegmentedControl.widthAnchor.constraint(equalTo:
inputsContainerView.widthAnchor).isActive = true

loginRegisterSegmentedControl.heightAnchor.constraint(equalToConstan
t: 30).isActive = true
    }


    //Create variables that can change the layout constraints of the
view
    //and that can be accessed outisde of setupInputsContainerView()
    var inputsContainerViewHeightAnchor: NSLayoutConstraint?
    var nameTextFieldHeightAnchor: NSLayoutConstraint?
    var emailTextFieldHeightAnchor: NSLayoutConstraint?
    var passwordTextFieldHeightAnchor: NSLayoutConstraint?


    //This sets the constraints of the subview so the program knows
how to display it.
    func setupInputsContainerView() {
        //Constraints: x, y, z, width, height
```

```
        inputsContainerView.centerXAnchor.constraint(equalTo:
view.centerXAnchor).isActive = true
        inputsContainerView.centerYAnchor.constraint(equalTo:
view.centerYAnchor).isActive = true
        inputsContainerView.widthAnchor.constraint(equalTo:
view.widthAnchor, constant: -24).isActive = true

        //We replaced the hard coded height anchor with a variable
that we can access outside
        //of inputsContainerView, which will allow us to change the
height.
        inputsContainerViewHeightAnchor =
inputsContainerView.heightAnchor.constraint(equalToConstant: 150)
        inputsContainerViewHeightAnchor?.isActive = true


        //Add subviews
        inputsContainerView.addSubview(nameTextField)
        inputsContainerView.addSubview(nameSeparatorView)
        inputsContainerView.addSubview(emailTextField)
        inputsContainerView.addSubview(emailSeparatorView)
        inputsContainerView.addSubview(passwordTextField)


        //Set up the constraints for nameTextField
        nameTextField.leftAnchor.constraint(equalTo:
inputsContainerView.leftAnchor, constant: 12).isActive = true
        nameTextField.topAnchor.constraint(equalTo:
inputsContainerView.topAnchor).isActive = true
        nameTextField.widthAnchor.constraint(equalTo:
inputsContainerView.widthAnchor).isActive = true
        //Place the height anchor wihin the variable
nameTextFieldHeightAnchor
        nameTextFieldHeightAnchor =
nameTextField.heightAnchor.constraint(equalTo:
inputsContainerView.heightAnchor, multiplier: 1/3)
        nameTextFieldHeightAnchor?.isActive = true

        //Set up the constraints for nameSeparatorView
        nameSeparatorView.leftAnchor.constraint(equalTo:
inputsContainerView.leftAnchor).isActive = true
        nameSeparatorView.topAnchor.constraint(equalTo:
nameTextField.bottomAnchor).isActive = true
        nameSeparatorView.widthAnchor.constraint(equalTo:
nameTextField.widthAnchor).isActive = true
        nameSeparatorView.heightAnchor.constraint(equalToConstant:
1).isActive = true


        //Set up the constraints for emailTextField
        emailTextField.leftAnchor.constraint(equalTo:
inputsContainerView.leftAnchor, constant: 12).isActive = true
        emailTextField.topAnchor.constraint(equalTo:
nameTextField.bottomAnchor).isActive = true
```

```
        emailTextField.widthAnchor.constraint(equalTo:
inputsContainerView.widthAnchor).isActive = true
        //Place the height anchor wihin the variable
emailTextFieldHeightAnchor
        emailTextFieldHeightAnchor =
emailTextField.heightAnchor.constraint(equalTo:
inputsContainerView.heightAnchor, multiplier: 1/3)
        emailTextFieldHeightAnchor?.isActive = true

        //Set up constraints for emailSeparatorView
        emailSeparatorView.leftAnchor.constraint(equalTo:
inputsContainerView.leftAnchor).isActive = true
        emailSeparatorView.topAnchor.constraint(equalTo:
emailTextField.bottomAnchor).isActive = true
        emailSeparatorView.widthAnchor.constraint(equalTo:
nameTextField.widthAnchor).isActive = true
        emailSeparatorView.heightAnchor.constraint(equalToConstant:
1).isActive = true


        //Set up constraints for passwordTextField
        passwordTextField.leftAnchor.constraint(equalTo:
inputsContainerView.leftAnchor, constant: 12).isActive = true
        passwordTextField.topAnchor.constraint(equalTo:
emailSeparatorView.bottomAnchor).isActive = true
        passwordTextField.widthAnchor.constraint(equalTo:
inputsContainerView.widthAnchor).isActive = true
        //Place the height anchor wihin the variable
paswordTextFieldHeightAnchor
        passwordTextFieldHeightAnchor =
passwordTextField.heightAnchor.constraint(equalTo:
inputsContainerView.heightAnchor, multiplier: 1/3)
        passwordTextFieldHeightAnchor?.isActive = true

    }


    //Function to setup all the constraints neccessary to give
loginRegisterButton dimensions and a position on the screen
    func setupLoginRegisterButton() {
        loginRegisterButton.centerXAnchor.constraint(equalTo:
view.centerXAnchor).isActive = true
        loginRegisterButton.topAnchor.constraint(equalTo:
inputsContainerView.bottomAnchor, constant: 12).isActive = true
        loginRegisterButton.widthAnchor.constraint(equalTo:
inputsContainerView.widthAnchor).isActive = true
        loginRegisterButton.heightAnchor.constraint(equalToConstant:
50).isActive = true
    }

}
```

GoalTrackerScreen.swift:

```
import UIKit
import Firebase

class GoalTrackerScreen: UITableViewController {

    let cellID = "cellID"

    var goals = [Goal]()


    override func viewDidLoad() {
        super.viewDidLoad()

        //Make the background colour white
        view.backgroundColor = .white

        title = "Goals"

        //Add a logout button to the left of the navigation bar
        navigationItem.leftBarButtonItem = UIBarButtonItem(title:
"Logout", style: .plain, target: self, action:
#selector(displayLoginRegisterScreen))

        //Add a "+" button to the top right of the navigation bar
        navigationItem.rightBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .add, target: self, action:
#selector(displayNewGoalScreen))

        //Register custom class "GoalCell" as the default cell class
        tableView.register(GoalCell.self, forCellReuseIdentifier:
cellID)

        checkIfUserIsLoggedIn()
        observeUserGoals()
    }


    @objc func displayNewGoalScreen() {
        let newGoalScreen = NewGoalScreen()
        let navigationControllerNewGoalScreen =
UINavigationController(rootViewController: newGoalScreen)
        present(navigationControllerNewGoalScreen, animated: true,
completion: nil)
    }


    func checkIfUserIsLoggedIn() {
        //Checks to see if the user is logged in already
        if Auth.auth().currentUser?.uid == nil {
            //If the user is not logged in, display the login page
            displayLoginRegisterScreen()
        }
```

```
    }


    //This function can be used to display the login in/register
screen
    @objc func displayLoginRegisterScreen() {
        //Log the user out of firebase
        logOutCurrentUser()

        //Instansiate LoginRegisterScreen
        let loginRegisterScreen = LoginRegisterScreen()

        //Set the display to fullscreen
        loginRegisterScreen.modalPresentationStyle = .fullScreen

        //Display the LoginRegisterScreen object
        present(loginRegisterScreen, animated: true, completion:
nil)
    }


    //This logs the current user out of Firebase
    func logOutCurrentUser() {
        do {
            try Auth.auth().signOut()
        } catch let logoutError {
            print(logoutError)
        }
    }


    //Retrieves all the user's current goals from Firebase
    func observeUserGoals() {

        //Checks the current user is logged in
        guard let uid = Auth.auth().currentUser?.uid else {
            return
        }

        //Sets up a reference to the user's personal node that
extends "User-Goals"
        let ref = Database.database().reference().child("User-
Goals").child(uid)

        //Observes all the goal IDs stored at that node.
        ref.observe(.childAdded, with: { (snapshot) in

            //Sets "goalID" equal to the unique ID of each goal
stored at "User-Goals"->uid
            let goalID = snapshot.key

            //Sets up another database reference to each goal's ID
but within "Goals"
```

```
            let goalReference =
Database.database().reference().child("Goals").child(goalID)

            //Retrieves the data for each goal, stores it as an
instance of "Goal", and then adds it
            //to "goals[]"
            goalReference.observe(.value, with: { (snapshot) in

                if let dictionary = snapshot.value as? [String:
AnyObject] {

                    let goal = Goal()

                    goal.name = dictionary["name"] as? String
                    goal.startDate = dictionary["startDate"] as?
String
                    goal.endDate = dictionary["endDate"] as? String
                    goal.accountabilityEmail =
dictionary["accountabilityEmail"] as? String
                    goal.goalID = goalID

                    self.checkGoalEndDate(goal: goal)
                    self.goals.append(goal)
                }

                //Refreshes the table view
                self.tableView.reloadData()

            }, withCancel: nil)

        }, withCancel: nil)

    }


    //This function will iterate throught the "goals" list and
compare each goal's end date to today's
    func checkGoalEndDate(goal: Goal) {

        let currentDate = NSDate() as Date
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "dd/MM/yyyy"

        guard let endDate = dateFormatter.date(from: goal.endDate!)
else {
            return
        }

        //Compare dates
        if currentDate > endDate {

            //Set up a reference to "Users" in Firebase
            let ref = Database.database().reference().child("Users")
```

```
            //Observe all the values stored at the "Users" node
            ref.observe(.value, with: { (snapshot) in

                //Unwrapps the snapshot as a Dictionary
                if let dictionary = snapshot.value as? [String:
AnyObject] {

                    //Itterates through the dictionary
                    for item in dictionary {

                        //Unwrapps the email attribute of the "goal"
argument
                        guard let goalEmail =
goal.accountabilityEmail else {
                            return
                        }

                        //Unwraps the email retrieved from Firebase
                        guard let retrievedEmail =
item.value["email"] as? String? else {
                            return
                        }

                        //Checks to see if the email from Firebase
matches the email stored as an attribute of "goal"
                        if goalEmail == retrievedEmail {

                            let message = "Hey, I've missed a
deadline! Chase me up!"
                            let peerID = item.key

                            //Sends a text to the goal's linked user
if the goal has passed it's deadline.
                            self.sendMessage(messageText: message,
recipientID: peerID)

                        }

                    }

                }

            }, withCancel: nil)

        }

    }



    //Removes a goal from Firebase
    func removeGoal(withID goalID: String) {

        //Unwrap the optional userID
```

```
        guard let userID = Auth.auth().currentUser?.uid else {
            return
        }

        //Create references to Firebase, one to the goal in the
"Goals" sub-tree and the other to the
        //goal in the "User-Goals" sub-tree
        let goalRef =
Database.database().reference().child("Goals").child(goalID)
        let userGoalRef =
Database.database().reference().child("User-
Goals").child(userID).child(goalID)

        //Removes the value specified in the "goalRef" reference
        goalRef.removeValue { (error, _) in

            //Catches and prints any errors
            if error != nil {
                print(error?.localizedDescription as Any)
            }

        }

        //Removes the value specified in the "userGoalRef" reference
        userGoalRef.removeValue { (error, _) in

            //Catches and prints any errors
            if error != nil {
                print(error?.localizedDescription as Any)
            }
        }
    }


    func sendMessage(messageText: String, recipientID: String) {

        //Creates a reference to a new node in firebase "Messages"
        let ref = Database.database().reference().child("Messages")
        //Creates a unique ID for this particular message node
        let childRef = ref.childByAutoId()
        //Uploads text and recipient from the text field to the
database.
        let toID = recipientID
        let fromID = Auth.auth().currentUser!.uid
        let timeStap = NSDate().timeIntervalSince1970
        let values = ["text": messageText, "toID": toID, "fromID":
fromID, "timeStamp": timeStap] as [String : Any]

        //Adds the  key-value pairs stored within the dictionary
"values" to the "childByAutoId" node
        //If those values are successfully added, the completion
block runs.
        childRef.updateChildValues(values) { (error, ref) in
```

```
            //Catches and prints any errors to the console
            if error != nil {
                print(error as Any)
            }

            guard let messageID = childRef.key else {
                return
            }

            //Sets up a reference to a node called "User-Messages"
and creates a chid node
            //Based on the user's ID
            let userMessageRef =
Database.database().reference().child("User-Messages").child(fromID)
            //Adds the message ID to user's ID node.
            userMessageRef.updateChildValues([messageID: true]) {
(error, ref) in
                print("Inside userMessageRef")
                if error != nil{
                    print(error as Any)
                    return
                }
            }

            //Creates a node in "User-Messages" based on the
recipients ID
            //Stores the message ID at that node.
            let recipientUserMessagesRef =
Database.database().reference().child("User-Messages").child(toID)
            recipientUserMessagesRef.updateChildValues([messageID:
true]) {(error, ref) in
                if error != nil {
                    print(error as Any)
                    return
                }
            }

        }
    }


    /* Setup and modify the table view */

    //Determine the number of rows in the "UITableView" equal to the
number of goals in "Goals"
    override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
        return goals.count
    }


    //Sets the information of each cell to match each goal
    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
```

```
        //Creates an instance of the GoalCell class for each value
of indexPath
        let cell = tableView.dequeueReusableCell(withIdentifier:
cellID, for: indexPath) as! GoalCell

        //Gets each goal from the goals list where the index of the
item fetched
        //matches the row number in the table
        let goal = goals[indexPath.row]
        //Sets the value of the "goal" attribute in "GoalCell" to
the "goal" constant
        cell.goal = goal

        return cell
    }


    //Make the cells in the table bigger
    override func tableView(_ tableView: UITableView, heightForRowAt
indexPath: IndexPath) -> CGFloat {
        return 72
    }


    /* "Swipe-to-delete" code: */

    //Allows the user to edit the table
    override func tableView(_ tableView: UITableView, canEditRowAt
indexPath: IndexPath) -> Bool {
        return true
    }


    //Configures the action taken when a use swipes a cell.
    override func tableView(_ tableView: UITableView,
trailingSwipeActionsConfigurationForRowAt indexPath: IndexPath) ->
UISwipeActionsConfiguration? {

        //Creates a new contextual action and set's its basic
attributes
        let deleteAction = UIContextualAction(style: .destructive,
title: "Complete") { (_, _, complete) in

            //Retrieves the "goalID" of the goal to be removed and
unwraps it.
            guard let goalID = self.goals[indexPath.row].goalID else
{
                return
            }

            //When the action is called, the "goalCell" swiped is
removed from the "goals" list, from the table, and from Firebase.
            self.goals.remove(at: indexPath.row)
```

```
            self.tableView.deleteRows(at: [indexPath], with:
.automatic)
            self.removeGoal(withID: goalID)

            complete(true)
        }


        //Changes the background colour of the delete button
        deleteAction.backgroundColor = UIColor(displayP3Red: 22/255,
green: 160/255, blue: 133/255, alpha: 1.0)

        //Sets "deleteAction" as the action taken when a cell is
swiped completely to the left or right.
        let configuration = UISwipeActionsConfiguration(actions:
[deleteAction])
        configuration.performsFirstActionWithFullSwipe = true

        return configuration
    }


}
```

NewGoalScreen.swift:

```
import UIKit
import Firebase

class NewGoalScreen: UIViewController {

    var goalPickerDate: String = ""

    /* Define all the UI elements */

    //Set up the "Goal Name" text field
    let goalNameTextField: UITextField = {
        let tf = UITextField()
        tf.placeholder = "Goal Name"
        tf.borderStyle = .roundedRect
        tf.translatesAutoresizingMaskIntoConstraints = false
        return tf
    }()


    //Set up the "Accountablility" text field
    let accountabilityTextField: UITextField = {
        let tf = UITextField()
        tf.placeholder = "Peer's Email"
        tf.borderStyle = .roundedRect
        tf.translatesAutoresizingMaskIntoConstraints = false
        return tf
```

```
    }()


    //Set up the UIDatePicker
    let goalEndDatePicker: UIDatePicker = {
        let dp = UIDatePicker()
        dp.datePickerMode = UIDatePicker.Mode.date
        //dp.minimumDate = NSDate() as Date
        dp.addTarget(self, action:
#selector(changeGoalPickerDate(datePicker:)), for: .valueChanged)
        dp.translatesAutoresizingMaskIntoConstraints = false
        return dp
    }()


    //Set up the submit button
    let submitButton: UIButton = {
        let bn = UIButton(type: .system)
        bn.backgroundColor = UIColor(displayP3Red: 22/255, green:
160/255, blue: 133/255, alpha: 1.0)
        bn.setTitleColor(UIColor.white, for: .normal)
        bn.setTitle("Create Goal", for: .normal)
        bn.layer.cornerRadius = 5
        bn.addTarget(self, action: #selector(createNewGoal), for:
.touchUpInside)
        bn.translatesAutoresizingMaskIntoConstraints = false
        return bn
    }()



    override func viewDidLoad() {
        super.viewDidLoad()

        //Set the background colour
        view.backgroundColor = .white

        //Set the view title
        title = "New Goal"

        //Adds a cancel button to the navigation bar
        navigationItem.leftBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .cancel, target: self, action:
#selector(handleCancel))

        //Adds all the subviews
        view.addSubview(goalNameTextField)
        view.addSubview(accountabilityTextField)
        view.addSubview(goalEndDatePicker)
        view.addSubview(submitButton)

        //Set up all UI constraints
        setupGoalNameTextField()
        setupAccountablilityTextField()
```

```
        setupGoalEndDatePicker()
        setupSubmitButton()
    }



    //Dismisses the view when called
    @objc func handleCancel() {
        self.dismiss(animated: true, completion: nil)
    }


    //Sets the end date of the goal based on the goal picker reading
    @objc func changeGoalPickerDate(datePicker: UIDatePicker) {
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "dd/MM/yyyy"
        goalPickerDate = dateFormatter.string(from: datePicker.date)
    }


    //Gets the current date and returns it as a string
    func getCurrentDate() -> String {
        let currentDate = Date()
        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "dd/MM/yyyy"
        let currentDateString = dateFormatter.string(from:
currentDate)
        return currentDateString
    }


    //Uploads the goal to Firebase and creates a reference for it
under "User-Goals"
    @objc func createNewGoal() {

        //Creates a reference to a node called "Goals" in Firebase
        let ref = Database.database().reference().child("Goals")

        //Sets up another reference, this time to an auto-generate
child node of "Goals"
        let childRef = ref.childByAutoId()

        //Get the values found in the various input UI elements
        let goalName = goalNameTextField.text!
        let email = accountabilityTextField.text!
        let startDate = getCurrentDate()
        let endDate = goalPickerDate

        //Store the user input as a dictionary
        let values = ["name": goalName, "accountabilityEmail":
email, "startDate": startDate, "endDate": endDate] as [String : Any]

        //Upload the user input to Firebase as a goal
        childRef.updateChildValues(values) { (error, ref) in
```

```
            //Checks for and catches any errors
            if error != nil{
                print(error as Any)
                return
            }

            //Unwraps goalID
            guard let goalID = childRef.key else {
                return
            }

            //Gets the current users unique Firebase ID
            let currentUID = Auth.auth().currentUser!.uid

            //Creates a new database reference to "currentUID" as a
sub-node of "User Goals"
            let userGoalRef =
Database.database().reference().child("User-
Goals").child(currentUID)

            //Uploads the goal ID as a sub-node of the user's ID
            userGoalRef.updateChildValues([goalID: true])
        }

        //Dismisses itself.
        handleCancel()
    }


    /* Defines the functions that will set up the UI constraints*/

    //Sets up the constraints for goalNameTextField
    func setupGoalNameTextField() {
        goalNameTextField.topAnchor.constraint(equalTo:
view.topAnchor, constant: 100).isActive = true
        goalNameTextField.centerXAnchor.constraint(equalTo:
view.centerXAnchor).isActive = true
        goalNameTextField.heightAnchor.constraint(equalToConstant:
50).isActive = true
        goalNameTextField.widthAnchor.constraint(equalTo:
view.widthAnchor, constant: -36).isActive = true
    }


    //Sets up th constraints for accountabilityTextField
    func setupAccountablilityTextField() {
        accountabilityTextField.topAnchor.constraint(equalTo:
view.topAnchor, constant: 200).isActive = true
        accountabilityTextField.centerXAnchor.constraint(equalTo:
view.centerXAnchor).isActive = true

accountabilityTextField.heightAnchor.constraint(equalToConstant:
50).isActive = true
```

```
        accountabilityTextField.widthAnchor.constraint(equalTo:
view.widthAnchor, constant: -36).isActive = true
    }


    //Sets up the constraints for goalEndDatePicker
    func setupGoalEndDatePicker() {
        goalEndDatePicker.topAnchor.constraint(equalTo:
view.topAnchor, constant: 250).isActive = true
        goalEndDatePicker.heightAnchor.constraint(equalToConstant:
300).isActive = true
        goalEndDatePicker.centerXAnchor.constraint(equalTo:
view.centerXAnchor).isActive = true
        goalEndDatePicker.widthAnchor.constraint(equalTo:
view.widthAnchor, constant: -50).isActive = true
    }


    //Sets up the constraints for the submit button
    func setupSubmitButton() {
        submitButton.bottomAnchor.constraint(equalTo:
view.bottomAnchor, constant: -50).isActive = true
        submitButton.widthAnchor.constraint(equalTo:
view.widthAnchor, constant: -72).isActive = true
        submitButton.heightAnchor.constraint(equalToConstant:
50).isActive = true
        submitButton.centerXAnchor.constraint(equalTo:
view.centerXAnchor).isActive = true
    }

}
```

ChatScreen.swift

```
import UIKit
import Firebase

class ChatScreen: UITableViewController {

    var messages = [Message]()
    var messagesDictionary = [String: Message]()

    let cellID = "cellID"

    override func viewDidLoad() {
        super.viewDidLoad()

        view.backgroundColor = .white

        title = "Chats"

        //Add a logout button to the left of the navigation bar
```

```
        navigationItem.leftBarButtonItem = UIBarButtonItem(title:
"Logout", style: .plain, target: self, action:
#selector(displayLoginRegisterScreen))

        navigationItem.rightBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .compose, target: self, action:
#selector(handleNewMessage))

        //Registers the custom cell type UserCell with the
tableView.
        tableView.register(UserCell.self, forCellReuseIdentifier:
cellID)

        observeUserMessages()

    }

    //Fetches the messages from Firebase, relevant to the current
user, to be displayed.
    func observeUserMessages() {
        guard let uid = Auth.auth().currentUser?.uid else {
            return
        }

        let ref = Database.database().reference().child("User-
Messages").child(uid)

        ref.observe(.childAdded, with: {(snapshot) in

            let messageID = snapshot.key
            let messagesReference =
Database.database().reference().child("Messages").child(messageID)

            messagesReference.observe(.value, with: { (snapshot) in

                //Stores each message as a Message()
                if let dictionary = snapshot.value as? [String:
AnyObject] {
                    let message = Message()

                    message.fromID = dictionary["fromID"] as!
String?
                    message.toID = dictionary["toID"] as! String?
                    message.text = dictionary["text"] as! String?
                    message.timeStamp = dictionary["timeStamp"] as!
NSNumber?

                    //Checks if the message has a toID attribute
                    //If it does, it creates a key-value pair in the
messagesDictionary
                    //where the ID is the key and the message is the
value.
                    //If the toID already exists in the dictionary,
it's value gets updated to the
```

```
                    //text of the latest message sent.
                    if let chatPartnerID = message.chatPartnerID() {
                        self.messagesDictionary[chatPartnerID] =
message

                        self.messages =
Array(self.messagesDictionary.values)

                        //Sorts the different chats so the most
recently active is
                        //displayed at the top of the table.
                        self.messages.sort { (message1, message2) ->
Bool in
                            return message1.timeStamp!.intValue >
message2.timeStamp!.intValue
                        }
                    }

                    //Reloads the table with the new data
                    self.tableView.reloadData()
                }

            }, withCancel: nil)

        }, withCancel: nil)
    }


    //Sets the number of rows in the table to the number of messages
    override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
        return messages.count
    }


    //Sets the title of each cell to the text of the message.
    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {

        //Creates an instance of the UserCell class for each value
of indexPath
        let cell = tableView.dequeueReusableCell(withIdentifier:
cellID, for: indexPath) as! UserCell

        //Gets each message from the messages list where the index
of the item fetched
        //matches the row number in the table
        let message = messages[indexPath.row]
        //Sets the value of attribute in UserCell to the message
constant
        cell.message = message

        return cell
    }
```

```
    //Make the cells in the table bigger
    override func tableView(_ tableView: UITableView, heightForRowAt
indexPath: IndexPath) -> CGFloat {
        return 72
    }


    //Displays the given user's chat screen when their cell is
selected in the table view
    override func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {

        //Find the message corresponding to the cell selected
        let message = messages[indexPath.row]

        //Get's the ID of the other user in the chat by calling the
"chatPartnerID" method.
        guard let chatPartnerID = message.chatPartnerID() else {
            return
        }

        //Sets up a reference to the aforementioned user's node
        let ref =
Database.database().reference().child("Users").child(chatPartnerID)

        //Observes the values at the user's node
        ref.observeSingleEvent(of: .value, with: { (snapshot) in

            //Creates a dictionary to store the observed values
            guard let dictionary = snapshot.value as? [String:
AnyObject] else {
                return
            }

            //Creates a user object
            let user = User()

            //Sets the attribute of the object based on the data
stored in the dictionary
            user.name = dictionary["name"] as? String
            user.email = dictionary["email"] as? String
            user.id = snapshot.key

            //Display's the "chatLogContoller" for the selected
user.
            self.displayChatLogControllerForUser(user: user)

        }, withCancel: nil)
    }


    //Function that will display chatLogController when called.
```

```
    func displayChatLogControllerForUser(user: User) {
        let chatLogController =
ChatLogController(collectionViewLayout:
UICollectionViewFlowLayout())
        chatLogController.hidesBottomBarWhenPushed = true
        //Sets the attribute "user" of chatLogController to the
argument "user".
        chatLogController.user = user
        navigationController?.pushViewController(chatLogController,
animated: true)
    }


    //This function can be used to display the login in/register
screen
    @objc func displayLoginRegisterScreen() {
        //Log the user out of firebase
        logOutCurrentUser()

        //Instansiate LoginRegisterScreen
        let loginRegisterScreen = LoginRegisterScreen()

        //Set the display to fullscreen
        loginRegisterScreen.modalPresentationStyle = .fullScreen

        //Display the LoginRegisterScreen object
        present(loginRegisterScreen, animated: true, completion:
nil)
    }


    //This function will display an instance of the
UINavigationController class within a
    //root view controller that is an instance of NewMessageScreen.
    @objc func handleNewMessage() {
        let newMessageScreen = NewMessageScreen()
        newMessageScreen.chatScreen = self
        let navigationController =
UINavigationController(rootViewController: newMessageScreen)
        present(navigationController, animated: true, completion:
nil)
    }


    //This logs the current user out of Firebase
    func logOutCurrentUser() {
        do {
            try Auth.auth().signOut()
        } catch let logoutError {
            print(logoutError)
        }
    }
```

```
    //Gets the current user's information when called
    func getCurrentUsersInfo() {
        let uid = Auth.auth().currentUser?.uid

Database.database().reference().child("Users").child(uid!).observeSi
ngleEvent(of: .value, with: { (snapshot) in

            print(snapshot)

        }, withCancel: nil)

    }
}
```

ChatLogController.swift:

```
import UIKit
import Firebase

class ChatLogController: UICollectionViewController,
UITextFieldDelegate, UICollectionViewDelegateFlowLayout {


    //Sets the title of the instance of ChatLogController to the
selected user's name
    //as soon as the user's name is set.
    var user: User? {
        didSet {
            navigationItem.title = user?.name

            //Calls observe messages
            observeMessages()
        }
    }


    //Stores the messages of the current user.
    var messages = [Message]()


    //Observes the messages of which the current user is either the
sender or recipient and appends them to the
    //"messages" list.
    func observeMessages() {

        //Unwrapps the current user UID
        guard let uid = Auth.auth().currentUser?.uid else {
            return
        }

        //Sets up a reference to the "User-Messages" node in
Firebase
```

```
        let userMessagesRef =
Database.database().reference().child("User-Messages").child(uid)

        //Observes each message at the current user's UID sub-node
        userMessagesRef.observe(.childAdded, with: { (snapshot) in

            //Gets the message ID as the snapshot key
            let messageID = snapshot.key

            //Sets up another reference to the "Messages" node this
time
            let messagesRef =
Database.database().reference().child("Messages").child(messageID)

            //Observes each message either sent or recieved by the
current user
            messagesRef.observe(.value, with: { (snapshot) in

                //Unwrapps the values found and stores them in a
dictionary
                guard let dictionary = snapshot.value as? [String:
AnyObject] else {
                    return
                }

                //Creates a message object for each message found
                let message = Message()

                //Sets the "message" attributes to the corresponding
values from the database
                message.text = dictionary["text"] as? String
                message.fromID = dictionary["fromID"] as? String
                message.timeStamp = dictionary["timeStamp"] as?
NSNumber
                message.toID = dictionary["toID"] as? String

                //Makes sure only messages that belong in that chat
are displayed
                if message.chatPartnerID() == self.user?.id {
                    //Adds the each "message" to the "messages"
array
                    self.messages.append(message)

                    //Reloads the table data to display all the
messages.
                    self.collectionView.reloadData()
                }

            }, withCancel: nil)

        }, withCancel: nil)

    }
```

```
    //This reason we define inputTextField in this way is so that we
can
    //reference it outside of the setupInputComponents() function
    lazy var inputTextField: UITextField = {
        let textField = UITextField()
        textField.placeholder = "Enter message..."
        textField.translatesAutoresizingMaskIntoConstraints = false
        textField.delegate = self
        return textField
    }()


    //Create a cell ID
    let cellID = "cellID"

    override func viewDidLoad() {
        super.viewDidLoad()

        //Sets the background colour of the UIView at the bottom of
the screen to solid white
        //This makes sure you cannot see the messages through the
text bar.
        //Also gives the top message some padding from the
navigation bar
        collectionView?.contentInset = UIEdgeInsets(top: 10, left:
0, bottom: 58, right: 0)
        collectionView?.alwaysBounceVertical = true
        collectionView?.backgroundColor = .white
        collectionView?.register(ChatMessageCell.self,
forCellWithReuseIdentifier: cellID)

        setupInputComponents()
    }


    //Set the size of each cell being displayed
    func collectionView(_ collectionView: UICollectionView, layout
collectionViewLayout: UICollectionViewLayout, sizeForItemAt
indexPath: IndexPath) -> CGSize {

        //Creates a height variable for the cell and gives it a base
value
        var height: CGFloat = 80

        //Checks the message has text
        if let text = messages[indexPath.item].text {
            height = estimateFrameForText(text: text).height + 20
        }

        return CGSize(width: view.frame.width, height: height)
    }
```

```
    //Gets an estimate for the size of a message cell based off of
the text to be displayed.
    private func estimateFrameForText(text: String) -> CGRect {

        let size = CGSize(width: 200, height: 1000)
        let options =
NSStringDrawingOptions.usesFontLeading.union(.usesLineFragmentOrigin
)

        return NSString(string: text).boundingRect(with: size,
options: options, attributes: [NSAttributedString.Key.font :
UIFont.systemFont(ofSize: 16)], context: nil)
    }


    //Sets the number of cells in the collection view
    override func collectionView(_ collectionView: UICollectionView,
numberOfItemsInSection section: Int) -> Int {
        //Sets the number of cells to be displayed equal to the
number of messages in "messages".
        return messages.count
    }


    //Controlls the content of each cell
    override func collectionView(_ collectionView: UICollectionView,
cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {

        //Creates a cell as an instance of "ChatMessageCell"
        let cell =
collectionView.dequeueReusableCell(withReuseIdentifier: cellID, for:
indexPath) as! ChatMessageCell

        //Retrieves the text of each message in the "messages" list
        let message = messages[indexPath.item]

        //Sets the "text" attribute of each cell equal to the text
of one of the messages.
        cell.textView.text = message.text

        //Modify the message buble width
        cell.bubbleWidthAnchor?.constant =
estimateFrameForText(text: message.text!).width + 32

        return cell
    }


    //Setup the GUI
    func setupInputComponents() {
        //Set up container at the bottom of the screen
        let containerView = UIView()
        containerView.backgroundColor = .white
```

```
        containerView.translatesAutoresizingMaskIntoConstraints =
false

        view.addSubview(containerView)

        //Constraints for the container
        containerView.leftAnchor.constraint(equalTo:
view.leftAnchor).isActive = true
        containerView.widthAnchor.constraint(equalTo:
view.widthAnchor).isActive = true
        containerView.bottomAnchor.constraint(equalTo:
view.bottomAnchor).isActive = true
        containerView.heightAnchor.constraint(equalToConstant:
83).isActive = true


        //Setup send button
        let sendButton = UIButton(type: .system)
        sendButton.setTitle("Send", for: .normal)
        sendButton.translatesAutoresizingMaskIntoConstraints = false
        sendButton.addTarget(self, action: #selector(handleSend),
for: .touchUpInside)

        containerView.addSubview(sendButton)

        //Constrain sendButton
        sendButton.rightAnchor.constraint(equalTo:
containerView.rightAnchor).isActive = true
        sendButton.centerYAnchor.constraint(equalTo:
containerView.centerYAnchor).isActive = true
        sendButton.widthAnchor.constraint(equalToConstant:
80).isActive = true
        sendButton.heightAnchor.constraint(equalTo:
containerView.heightAnchor).isActive = true


        //Add inputTextField to subview.
        containerView.addSubview(inputTextField)

        //Constraints for inputTextField
        inputTextField.leftAnchor.constraint(equalTo:
containerView.leftAnchor, constant: 8).isActive = true
        inputTextField.centerYAnchor.constraint(equalTo:
containerView.centerYAnchor).isActive = true
        inputTextField.rightAnchor.constraint(equalTo:
sendButton.leftAnchor).isActive = true
        inputTextField.heightAnchor.constraint(equalTo:
containerView.heightAnchor).isActive = true


        //Create separator to seperate the containerView from the
rest of view
        let separatorLineView = UIView()
```

```
        separatorLineView.backgroundColor = UIColor(displayP3Red:
220/255, green: 220/255, blue: 220/255, alpha: 1)
        separatorLineView.translatesAutoresizingMaskIntoConstraints
= false

        containerView.addSubview(separatorLineView)

        //Constrain separatorLineView
        separatorLineView.leftAnchor.constraint(equalTo:
containerView.leftAnchor).isActive = true
        separatorLineView.widthAnchor.constraint(equalTo:
containerView.widthAnchor).isActive = true
        separatorLineView.topAnchor.constraint(equalTo:
containerView.topAnchor).isActive = true
        separatorLineView.heightAnchor.constraint(equalToConstant:
1).isActive = true

    }


    //Uploads the text from the inputTextField to the cloud when
called
    @objc func handleSend() {
        //Creates a reference to a new node in firebase "Messages"
        let ref = Database.database().reference().child("Messages")
        //Creates a unique ID for this particular message node
        let childRef = ref.childByAutoId()
        //Uploads text and recipient from the text field to the
database.
        let toID = user!.id!
        let fromID = Auth.auth().currentUser!.uid
        let timeStap = NSDate().timeIntervalSince1970
        let values = ["text": inputTextField.text!, "toID": toID,
"fromID": fromID, "timeStamp": timeStap] as [String : Any]

        //Adds the  key-value pairs stored within the dictionary
"values" to the "childByAutoId" node
        //If those values are successfully added, the completion
block runs.
        childRef.updateChildValues(values) { (error, ref) in

            //Catches and prints any errors to the console
            if error != nil {
                print(error as Any)
            }

            guard let messageID = childRef.key else {
                return
            }

            //Sets up a reference to a node called "User-Messages"
and creates a chid node
            //Based on the user's ID
```

```
            let userMessageRef =
Database.database().reference().child("User-Messages").child(fromID)
            //Adds the message ID to user's ID node.
            userMessageRef.updateChildValues([messageID: true]) {
(error, ref) in
                print("Inside userMessageRef")
                if error != nil{
                    print(error as Any)
                    return
                }
            }

            //Creates a node in "User-Messages" based on the
recipients ID
            //Stores the message ID at that node.
            let recipientUserMessagesRef =
Database.database().reference().child("User-Messages").child(toID)
            recipientUserMessagesRef.updateChildValues([messageID:
true]) {(error, ref) in
                if error != nil {
                    print(error as Any)
                    return
                }
            }

        }
        //Clears the text field
        inputTextField.text = ""
    }


    //Allows the user to send a message by pressing the "Enter" key
    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        handleSend()
        return true
    }
}
```

NewMessageScreen.swift:

```
import UIKit
import Firebase

class NewMessageScreen: UITableViewController {

    let cellID = "cellID"

    var users = [User]()

    override func viewDidLoad() {
        super.viewDidLoad()
```

```
        //Adds the cancel button to the navigation bar
        navigationItem.leftBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .cancel, target: self, action:
#selector(handleCancel))

        //Changes the title of the view controller
        title = "New Message"

        //Registers our custom cell class.
        tableView.register(UserCell.self, forCellReuseIdentifier:
cellID)

        //Fetch the users from Firebase
        fetchUsers()

    }


    //When run, fetches users from firebase
    func fetchUsers() {

        //Create a Firebase reference

Database.database().reference().child("Users").observe(.childAdded,
with: { (snapshot) in

            //Creates a variable "dictionary" and stores the values
contained within the snapshot as a
            //key-value pair within the dictionary data structure.
            if let dictionary = snapshot.value as? [String:
AnyObject] {
                //Creates an instance of User
                let user = User()

                //Gives the attributes of user values
                user.name = dictionary["name"] as? String
                user.email = dictionary["email"] as? String
                user.id = snapshot.key

                //Adds the new user to the list.
                self.users.append(user)

                self.tableView.reloadData()
            }


        }, withCancel: nil)

    }


    //Dismisses the screen when the user presses the cancel button.
    @objc func handleCancel() {
        self.dismiss(animated: true, completion: nil)
```

```
    }


    //Overides the tableView function from the UITableViewController
parent class and sets
    //the number of rows to be displayed with information to the
number of users.
    override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
        return users.count
    }


    //A custom cell for each row we are displaying and displays each
user's name and email on the relevant cell.
    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {

        let cell = tableView.dequeueReusableCell(withIdentifier:
cellID, for: indexPath)

        let user = users[indexPath.row]

        cell.textLabel?.text = user.name
        cell.detailTextLabel?.text = user.email

        return cell
    }


    //Makes the cells a bit taller
    override func tableView(_ tableView: UITableView, heightForRowAt
indexPath: IndexPath) -> CGFloat {
        return 52
    }

    //Create an instance of chat screen so we can access the
function displayChatLogController
    var chatScreen: ChatScreen?

    //Instruct the app to call displayChatLogController() when a
cell is selected.
    override func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {
        dismiss(animated: true, completion: {
            //Sets the constant user equal to the user selected from
the contacts list.
            let user = self.users[indexPath.row]
            //Calls displayChatLogControllerForUser() for the
current selected user
            self.chatScreen?.displayChatLogControllerForUser(user:
user)
        })
    }
```

```
}
```

UserCell.swift:

```
import UIKit
import Firebase

class UserCell: UITableViewCell {

    var message: Message? {
        didSet {

            setupName()

            //Sets the subtext of the cell equal to the contents of
the message.
            self.detailTextLabel?.text = message?.text
            self.detailTextLabel?.textColor = UIColor.darkGray

            //Check that the current message has a time stamp
            //If it does, format the timestamp to a date and set the
text of timeLabel
            //equal to the date.
            if let seconds = message?.timeStamp?.doubleValue {
                let timeStampDate = NSDate(timeIntervalSince1970:
seconds)

                let dateFormatter = DateFormatter()
                dateFormatter.dateFormat = "hh:mm:ss a"
                timeLabel.text = dateFormatter.string(from:
timeStampDate as Date)

            }
        }
    }


    private func setupName() {

        if let id = message?.chatPartnerID() {
            //Creates a reference to the message node with a
matching toID value.
            let ref =
Database.database().reference().child("Users").child(id)

            //Takes a snapshot of the values that extend the message
ID.
            ref.observeSingleEvent(of: .value, with: {(snapshot) in

                if let dictionary = snapshot.value as? [String:
AnyObject] {
```

```
                    //Sets the value of the cell's text to the value
stored at the child node "name".
                    self.textLabel?.text = dictionary["name"] as?
String

                }

            })
        }
    }


    //Creates a time stap at the edge of the message cell
    let timeLabel: UILabel = {
        let label = UILabel()
        label.font = UIFont.systemFont(ofSize: 13)
        label.textColor = UIColor.lightGray
        label.translatesAutoresizingMaskIntoConstraints = false
        return label
    }()


    //Changes the default style of the cell.
    override init(style: UITableViewCell.CellStyle, reuseIdentifier:
String?) {
        super.init(style: .subtitle, reuseIdentifier:
reuseIdentifier)
        addSubview(timeLabel)

        //Constraints for timeLabel
        timeLabel.rightAnchor.constraint(equalTo:
self.rightAnchor).isActive = true
        timeLabel.topAnchor.constraint(equalTo: self.topAnchor,
constant: 20).isActive = true
        //timeLabel.centerYAnchor.constraint(equalTo:
textLabel!.centerYAnchor).isActive = true
        timeLabel.widthAnchor.constraint(equalToConstant:
100).isActive = true
        timeLabel.heightAnchor.constraint(equalTo:
textLabel!.heightAnchor).isActive = true
    }


    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

GoalCell.swift

```
import UIKit
```

```
import Firebase

class GoalCell: UITableViewCell {

    //Give GoalCell the attribute "goal" of type "Goal?"
    var goal: Goal? {

        //"didSet" will run as soon as "goal" is assigned a value
        didSet {

            //Sets the text of the cell to the goal's name
            self.textLabel?.text = goal?.name

            //Unwraps the endDate property of goal
            guard let endDate = goal?.endDate else{
                return
            }

            //Sets the subtext of the cell to the end date and
alters its colour
            self.detailTextLabel?.text = "End date: " + endDate
            self.detailTextLabel?.textColor = UIColor.darkGray
        }
    }


    //Changes the default style of the cell.
    override init(style: UITableViewCell.CellStyle, reuseIdentifier:
String?) {
        super.init(style: .subtitle, reuseIdentifier:
reuseIdentifier)

    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

}
```

ChatMessageCell.swift

```
import UIKit

class ChatMessageCell: UICollectionViewCell {

    //Creates an attribute "textView" for the class that will allow
me to display message text
    let textView: UITextView = {
        let tv = UITextView()
        tv.text = "Sample text for now"
        tv.font = UIFont.systemFont(ofSize: 16)
```

```
        tv.translatesAutoresizingMaskIntoConstraints = false
        tv.backgroundColor = .clear
        tv.textColor = .white
        return tv
    }()

    //Create the background text bubble for each message.
    let bubbleView: UIView = {
        let view = UIView()
        view.backgroundColor = UIColor(displayP3Red: 22/255, green:
160/255, blue: 133/255, alpha: 1.0)
        view.translatesAutoresizingMaskIntoConstraints = false
        view.layer.cornerRadius = 16
        view.layer.masksToBounds = true
        return view
    }()


    //Set up a variable that will allow me to access and modify the
bubbleView's width externally
    var bubbleWidthAnchor: NSLayoutConstraint?


    //Override the constructor function
    override init(frame: CGRect) {
        //Inherit "frame" from the parent class
        super.init(frame: frame)

        addSubview(bubbleView)
        addSubview(textView)

        //Set up constraints for the "bubbleView"
        bubbleView.rightAnchor.constraint(equalTo: self.rightAnchor,
constant: -10).isActive = true
        bubbleView.topAnchor.constraint(equalTo:
self.topAnchor).isActive = true
        bubbleWidthAnchor =
bubbleView.widthAnchor.constraint(equalToConstant: 200)
        bubbleWidthAnchor?.isActive = true
        bubbleView.heightAnchor.constraint(equalTo:
self.heightAnchor).isActive = true

        //Set up the constraints for "textView"
        textView.leftAnchor.constraint(equalTo:
bubbleView.leftAnchor, constant: 8).isActive = true
        textView.topAnchor.constraint(equalTo:
self.topAnchor).isActive = true
        textView.rightAnchor.constraint(equalTo:
bubbleView.rightAnchor).isActive = true
        textView.heightAnchor.constraint(equalTo:
self.heightAnchor).isActive = true

    }
```

```
    //This is an extra peice of code needed to make the above
"override init" work.
    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

}
```

User.swift:

```
import UIKit

class User: NSObject {

    var id: String?
    var name: String?
    var email: String?

}
```

Message.swift:

```
import UIKit
import Firebase

class Message: NSObject {

    var fromID: String?
    var text: String?
    var timeStamp: NSNumber?
    var toID: String?


    func chatPartnerID() -> String? {

        return fromID == Auth.auth().currentUser?.uid ? toID :
fromID

    }

}
```

Goal.swift:

```
import UIKit

class Goal: NSObject {

    var goalID: String?
```

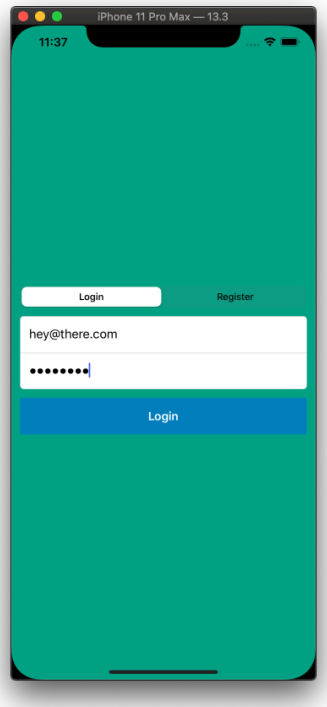
```
    var name: String?
    var accountabilityEmail: String?
    var startDate: String?
    var endDate: String?

}
```

Evaluation

Testing

Trying to login to an account that doesn't exist

Below I've entered details for an account that doesn't exist to see how the app handles it:



Expected response: The app will do nothing

Actual response: The app did nothing

If you try and login with credentials that are not on the system, the app will leave you on the login page.

For future development: Add an error message telling the user the account doesn't exist.

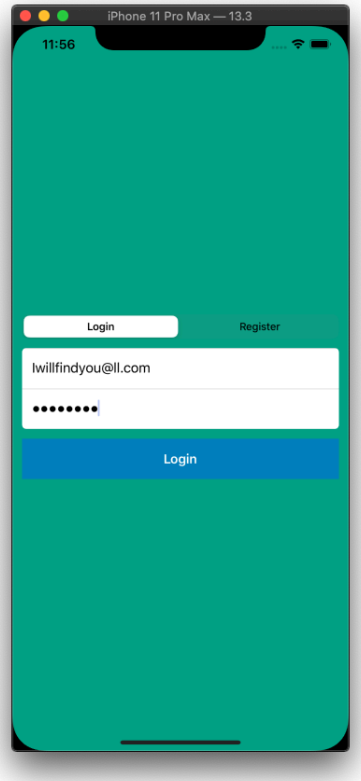
Trying to create an account with an invalid email

Here I'm entering a variety of invalid emails to check the app's response.

Email	Invalid because…	Expected Response	Actual Response
Heythere.com	Missing "@"	Stays on page	Stays on page and prints a format error to the console
Hey@there.somedomain	Top-level domain doesn't exist	Stays on page	Stays on page and prints a format error to the console
Josh@domain.com	Email already in use	Stays on page	Stays on page and prints an "email already in use error"

For future development: Add an error screen that pops up and lets the user know the reason their email is not being accepted.

Successfully logging in



Above, I've entered the correct information for John Wick's account and have successfully logged in.

Expected response: the application would display the Goals screen with Mr Wick's goals.

Actual response: the application displayed the Goals screen with the goals of the last logged in account.

The only way I can get the app to display Mr Wick's goals is by closing the app and re-running it.

For future development: refresh the Goals screen every time a new user is logged in.

Adding Goals

Goal Name	**Peer's Email**	**Date**	**Error**	**Expected Response**	**Actual Response**
Climb Snowdon	Josh.com	24 March 2020	Invalid email	Stays on page	Creates goal
Climb Snowdon	Josh@domain.com	23 February 2020	Date is earlier than the current date	Moves the date selector to the current date	Moves the date selector to the current date
	Josh@domain.com	24 March 2020	No goal name	Stays on page	Creates a nameless goal
Climb Snowdon		24 March 2020	No email	Stays on page	Creates a goal with no linked email and no date

This testing has revealed a few things:

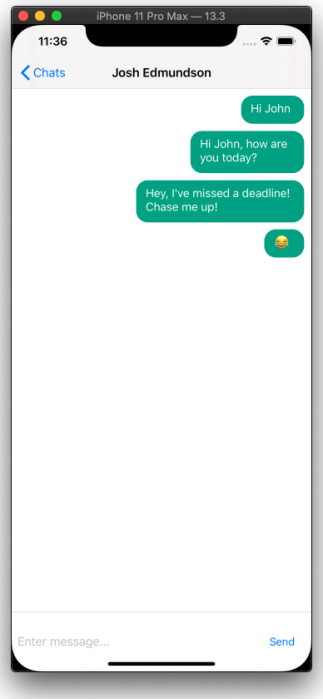
1. There is no provision in place for if the user enters an email not in the database.
2. The end date of a goal is only set if the value of the date picker is changed, thus if a user simply leaves the date picker on the current date, the goal will not be given a deadline.

For future development:

1. Check if the user enters anything into the email text field. If they do, check it's in the database and display an error message if it isn't.
2. Set the default deadline date of any goal to the current date.

Sending messages

The first thing I've tested is how the app handles sending data that is not text:

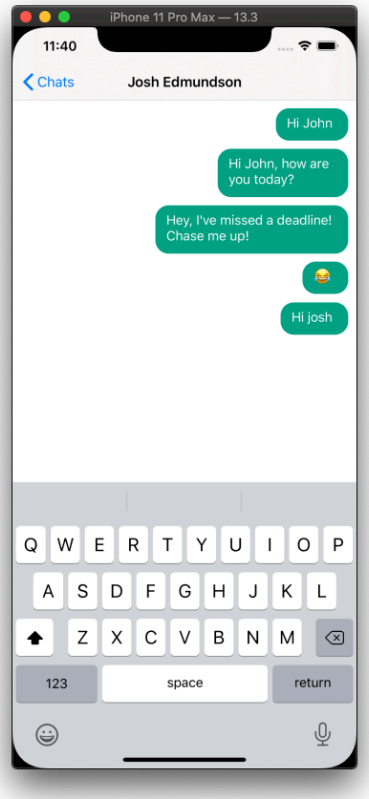


Which it seems to do without issue.

One problem with sending and receiving messages is the app makes no distinction between messages sent and messages received; they are both displayed in the same way. In the above image some of the messages displayed were sent by John to Josh and others from Josh to John yet they all look as though John sent them.

For future development: Identify whether a message has been sent or received and display them accordingly.

Another issue arises when I try and use the simulator's digital keyboard:

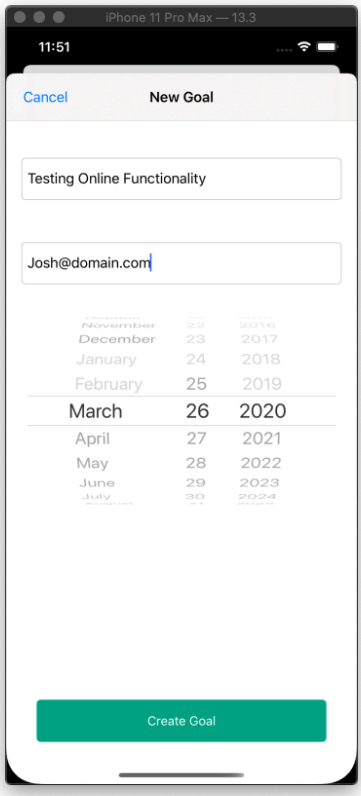


It completely hides the input text field from view.

For future development: Move the screen up with the digital keyboard.

Online Functionality

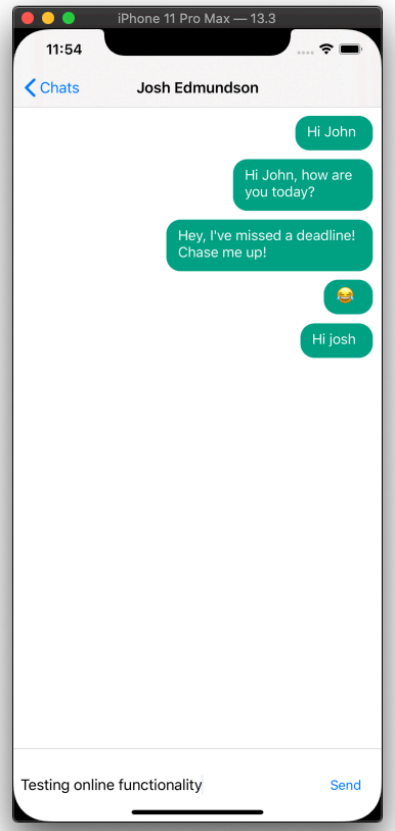
Adding goals:



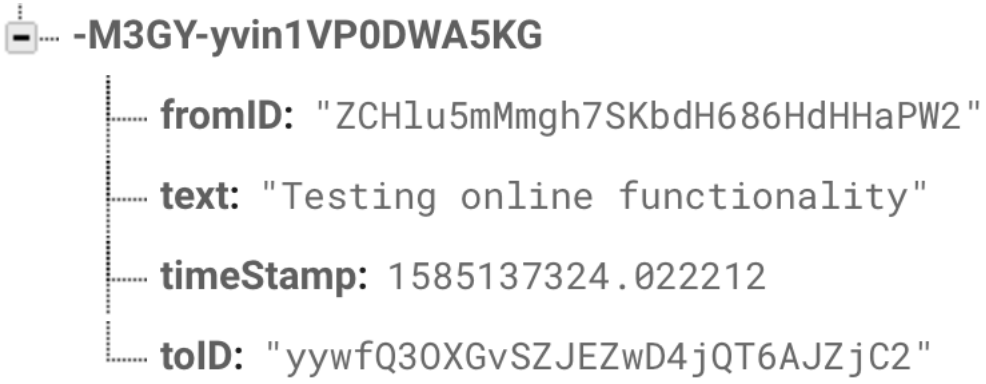
If I click “Create Goal”:

A goal is successfully created under the “Goals” node and a reference is added under “User-Goals”.

Sending messages:



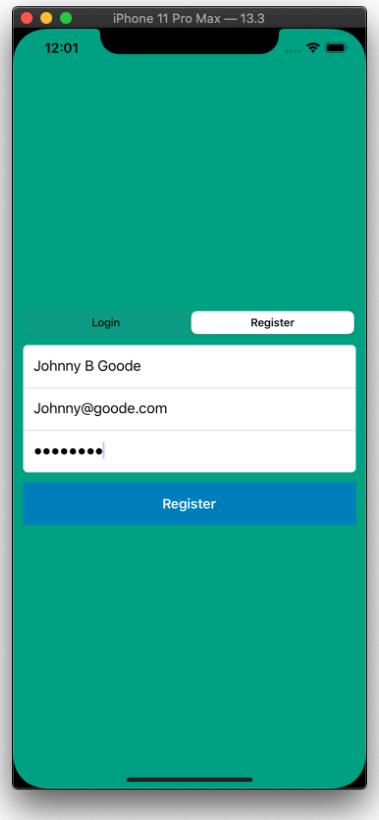
If I click send:



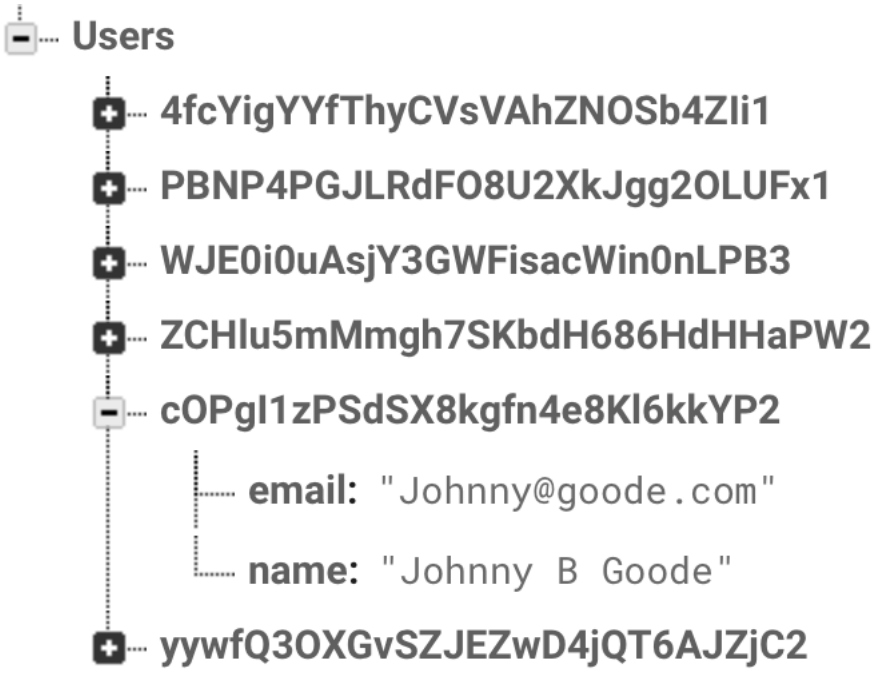
A message is successfully created under “Messages”

And successfully added under “User-Messages”

Creating new users:



If I click “Register”:



We can see a new user is successfully added.

Most of these online features and others (such as retrieving data from the online database) have been tested during development. You can see these tests in the "Development" section of this project.

Cross referencing tests with the success criteria

The success criteria I stated at the beginning of this project were:

- The user can create and track:
    - A goal
    - A habit (a regular short-term goal)
    - A project
- The user can successfully send and receive messages to and from a group chat
- The user can link a goal to an accountability group

In each of the above cases, I would say, based on the functionality of my app, that each criterium has only been partially met. Let's look at them one by one.

The user can create and track, a goal, a habit, and a project

During the "Analysis" component of this project I specified I wanted to be able to track three distinct types of goals which I each gave a name. A "goal" was a one-off event such as "get a promotion", something which you would not be aiming to do on a regular basis. A "habit" was a goal that would be repeated at regular intervals, for example the target "practice cello" could be set to recur daily, making it a "habit". A "project" was a "goal" with smaller sub-divisions that could each be achieved and accomplished. For example, a project might be "get an A* in A level computer science" which contains sub-goals such as, "Complete computer science project" and "make flashcards on data structures".

Based on my project, I have only managed to successfully implement one of the above features. Using this app, you can create and track "goals" but nothing else.

In future development, I could expand the goal the "New Goal" creator to include a selector where you choose the kind of goal you want to make, and which then changes the layout of the "New Goal" screen accordingly.

The user can successfully send and receive messages to and from a group chat

Again, here this criterium has been partially but not completely met. I have demonstrated the application's ability to send and receive messages between two people but there is currently nothing in place to support a multiple-user group chat.

In future development, I could implement a “Create Group” feature located in the “NewMessageScreen” that allows a user to create a group chat.

The user can link a goal to an accountability group

Similar to the aforementioned success criterium, this has only been partially met due to a lack of group-chat support. The user can successfully link goals to any one individual, but they can’t link a goal to multiple users.

In future development, I would change the “New Goal” creator to allow a user to link a chat ID instead of a user, allowing them to link one goal to a group chat.

Usability features

There were several things that popped up during testing that would need to be addressed in future development.

The software keyboard

In the testing component, I saw that the actual software-based keyboard covered the input text field when used. Obviously, this is not helpful for the user as they won’t be able to see what they are typing.

Displaying text messages

Again, in testing, I saw the app couldn’t distinguish between messages sent and received by the current user and simply displayed as thought they had all been sent. This makes it harder to distinguish who has said what in a conversation and makes it much more difficult to determine which user has failed to meet a goal deadline as the alert will appear to have come from whichever user is logged in.

In future development, this could be resolved by checking the ID of the user sending the message, comparing it to the ID of the user logged in, and then display the message accordingly.

Adding goals

There were a few usability issues when adding goals. If an email was entered that didn’t match any email in the online database, the user wasn’t alerted. In future development, I

could add some logic that would check the linked peer's email and cross-reference it with the emails stored in the database. The user would then be alerted if the email wasn't found.

Another issue was the fact a goal would be created without an end date unless the user changed the value of the date picker. In future development, I could add a default value to the end date attribute of each goal equal to the current date. That way, a user could leave the date picker on the current date and it would be set as the goal's end date.

Logging in and registering

Whilst the app was shown to be capable of identifying invalid input during testing, the user was not given any indication that something was wrong other than a lack of activity. In future development, I would check if any errors were flagged during the login/sign up process and, if there were any errors, display the appropriate message to alert the user as to what the issue is.

References

To construct the messaging component of this application, I largely followed an online course on constructing a chat application made by Brian Voong. Here is a link to the YouTube playlist:

https://www.youtube.com/playlist?list=PL0dzCUj1L5JEfHqwjBV0XFb9qx9cGXwkq