OCR Computer Science
Programming Project

# Countdown Widget

Bradley Richard Cable

# Table Of Contents


## Analysis 6

Introduction....6

Why solvable by a computer....6

*Non-computer examples*
*Why computers*

Stakeholders....7

*Who are they?*
*Examples of type of people / firms*
*Detail into examples*

Researching the market....8

*Stock iOS Calendar App*
*Third Party Apps*
*With all these options why make a new system?*

System and user requirements....11

Success Criteria....12

Limitations....13

*Sharing Countdowns*
*iCloud Backup / Sharing*
*Calendar Integration*

Survey with potential users....14

*Results*

## Design 17

Overview....17

Human Interface Guidelines....17

Looking into libraries and frameworks....17

*Database Storage Overview*
*Online mobile backend*
*Sign In With Apple*
*Local Database*
*iCloud Syncing (CloudKit)*

Decomposing the problem....22

*Algorithms Flowchart*

Designing some views ....27

*Main / Add event screen*
*Widget design*

# Development 30

General Overview of SwiftUI ....30

*Bit of history - UIKit and SwiftUI*
*Why SwiftUI is better and useful*

Starting Development ....31

Creating and Initialising SQLite ....31

*DatabaseConnector overview*
*Creating the database*
*Standardisation of event data*
*Create event function*
*Fetching Events*
*Testing database and its functions*

Main Screen Development ....38

*Card View (Subview)*
*How design works in SwiftUI*
*DateConverter*
*Card View (cont.)*
*Creating the list*
*Testing basic main screen*
*Relating back to the success criteria*

Event Creation Screen ....44

*NavigationView*
*The screen itself*
*Adapting the design*
*Date Picker*
*Colour Picker*
*Event Title*
*Putting it all together*

Add Event Testing ....52

*How I will display test data*
*Testing Date Picker*
*Fixing the same colour bug*
*Testing Text Input*
*Fixing the update issue*
*Relating back to the success criteria*

Widget Integration ....59

*WidgetKit*
*Sandboxes*
*Targets & Widgets*
*How does this relate to anything?*

*How I learned about everything above?*
*App Groups*
*IntentConfiguration*
*Timeline*
*TimelineEntry*
*Designing the widget*
*Widget Elements*
*The Provider*
*Intent*
*The Provider cont.*

Widget testing ..... 72

*Relating back to the success criteria*

Miscellaneous ..... 74

*Photo Background*
*Storing photos in DB*
*Added the photo to the widget*
*Testing widget feature*

User Feedback / Usability Testing ..... 79

Taking action ..... 82

Post testing & run-though ..... 86

*Adding Event*
*Widget System*

**Evaluation** **88**

Compare to success criteria ..... 88

*Completed Criteria - CC*
*What I missed - WIM*

Maintenance ..... 93

Limitations ..... 94

Releasing the app on the App Store ..... 95

**Comments** **99**

# Analysis

## Introduction

Calendars are a way of keeping track of events that are important in the coming weeks, months or even years. They organise these dates into a format which allows you see the events in relation to other events. This makes it very useful for a work setting, planning the days and weeks ahead seeing if you're busy or not. Not very fun, not very customisable.

In the same vain of calendar apps are the current countdown apps. Where users can start a countdown to a certain date. Allowing the user to keep track of events important to them in a clear concise way. For example: countdown to Christmas. But most aren't customisable.

In short I plan on creating a customisable countdown app which capitalises on the new features released in iOS 14 (September 16, 2020).

## Why solvable by a computer

Just like any problem people try and solve with technology, the end goal is to make it easier than other methods.

### *Non-computer examples*

- Write down the event on a calendar, work out how many days.
- Tear off calendars.
- Advent calendar candles.

All these examples above have similar problems / pitfalls compared to a computerised method:

A. Lots of mental arithmetic for some methods
B. Difficult if keeping track of multiple events
C. Waste of paper

### *Why computers*

So why is this problem solvable by a computer? One of the best things about computers is the ability to take in data, manipulate it and display it in some kind of digestible formate.

1. Input from the user, the event they want to keep track of.

2. Manipulating the date data with current date to workout value.

3. Display it in a clean, fun format.

For example, date formats in most high level programming languages have methods to allow calculations between two dates, as well as a way to formate it in a human readable string.

## Stakeholders

### *Who are they?*

A Stakeholder is anybody that can be interested or affected by something. With my app, the stakeholder range can be very general because everyone has events to countdown too (or used to anyway).

### *Examples of type of people / firms*

- Holidaymakers
- Concert goers
- People with bills / utility provider
- Students / Teachers
- Doctors / Patients

### *Detail into examples*

U*tility company and its customers:* depending on their plan, bills can be due at different times. This allows the customer to keep track of when they need to pay their bill, this also assures the utility company that bills are more likely to be paid on time. Following on with that. This could also apply to state bills, for example: self assessment tax returns, MOTs, benefit payout day, etc. Making sure you don't miss a tax return is vital

because it can lead to your business getting fined large amounts. Negligence is not an excuse.

Students / teachers: gives students the ability to organise their homework and due dates, especially for long projects (eg: this report). This can also give confidence to teachers that homework is going to be done on time. This would be even more applicable if a sharing feature was introduced to share countdowns with other users. Main user being a teacher and sharing with their students.

Finally, holidaymakers. Especially now in times of COVID, holiday makers are waiting for this to eventually end, and maybe a holiday in the coming months (hopefully) or next year. Especially with the customisation, for example, a holiday background (sandy beach). Will get them excited for the events coming up. Count it down day by day.

## Researching the market


### *Stock iOS Calendar App*

Firstly is there a built in stock iOS way of counting down to an event. Without downloading any apps. Can the built in calendar app do most of the job?

First off it is very convenient for the user as it is built into to iOS, for that exact reason it has mass support with integration into nearly all things that have anything to do with calendars using the iCalender (.ics) file format. This integration means other services can add/update/remove events with ease.

Obviously as it is built into the software to the operating system Apple do not limit the amount of events you can have, and put it behind a paywall. You can have as many events as you desire.

Furthermore it sends notifications when you want it to. Depending if you want one when the event is reached, one hour before, one day before, etc.

However all of this doesn't make it a good Countdown app. Same as a car doesn't make a good boat, it may have an engine but hell it doesn't float. These are the pitfalls on the app from a countdown perspective:

- Still the same problem with none computerised methods. You might know what date a certain event is on. However you would still have to calculate. How many days/weeks/months/years till that event happened. Doesn't take advantage of that fact its computerised.
- Limited customisation when it comes to events themselves. You can choose to organise them into different calendars with different colours, when to send notifications, etc. Apart from that very limited not personal whatsoever.
- Events that are long in the future, such as a summer holidays are hidden by the weekly or even daily view. Not allowing the user to get excited, see it tick from one month to the next.

All this sounds very petty, which is is. Nit picking a calendar app for not being a countdown. The Apple philosophy, minimalism is key. Anything complex needs to be abstracted away from the user so it looks like ✨magic🪄. Thats why there is so many 3rd Party apps. The calendar doesn't satisfy the users need to countdown events day be day. Even if it can store events to the minuet that isn't what a calendar is for.

***Third Party Apps***

There are many types of countdown apps current on the market. By on the market I mean the Apple App Store. I have picked one to look into. In this case I have chosen 'Countdown' by Mariano Rezk (link to app).

This apps features include:

- Create countdowns
- Pick any colour you'd like for your countdown background.
- Share your countdowns with friends and family.
- Notifications when events occur.

- Countdown repeats.
- Sync you're events across your devices with iCloud.

These are obviously the main features of the app. Copied from the App Store description. However I would like to point out these two specific features:

- Format in which you want to see your countdown
- Countdown Widget

These features stated above are ***premium only***, this app limits customisation. As well as a feature which I would argue is the most important of all, especially with this new iOS 14 update.

After looking though the main features, nearly all countdown have the same features as stated in the main section. These are necessary. You can't have a countdown app without the ability to make events. These features are vital for your average countdown app.

Comparing the stock app to this you can see there are a lot of similarities, creating events, notifications, repeating events, sync to iCloud, they are the same. They both have the engine (referring to my analogy above). I think the key difference is customisation.

***With all these options why make a new system?***

The key difference is customisation, that will be my focus. As well as making it free, no premium features, no in app purchases (that inhibit the main functionality). That will be my unique selling point. How I will allow my users too customise their events will be discussed more in the design section. Obviously bulks of the app will be similar to competition. As I would have to take date as input (etc)… At the end of the day its how you differ that matters. In this case my focus will be on customisation.

## System and user requirements

The system requirements of this app will be an iPhone running the latest major release of the iOS operating system (iOS 14). Luckily Apple has very high adoption rates of updates compared to competitors. At the time of writing (07/03/2020), the current device version rate *according to Apple* is 80% is on iOS 14. So In real terms, it isn't a requirement.

Obviously the app will only be available for iPhone as it will be coded in Swift, Apple's programming language. Therefore an iPhone is a requirement. Again luckily to Apple's image and popularity, the majority of the people 15-18 have iPhones.

Sort of linking to limitations, as I'm using Swift this will give me no option to port it onto other platforms such as Android, completely incompatible. Also due to time restraints I won't be learning Java just so it can be on Android.

## Success Criteria

*Widget System ~*

- Diffrent sized widget (.systemSmall, .systemMedium, etc)
- Customise the widget
  - Different colour text
  - Organise / Remove widget elements (count, date, name)
- Display image of choice as widget background
- Updates daily, as countdown decreases

*Event Creation ~*

- Input name of event
- Input date (time?) of event
- Input two colours which will create a gradient background
- Background image for event instead of gradient
- Repeating events (weekly, monthly, yearly)

*Miscellaneous ~*

- iCloud syncing with other devices
- Calendar integration
- iPad version

## Limitations

### *Sharing Countdowns*

Using the 3rd party app as a baseline, there are a few things I'm going remove or simplify. For example: "Share your countdowns with friends and family". I understand why this is vital feature for an app like this in the eyes of some users. Being able to share with friends makes the app. Firstly, gain popularity quicker, spreading though friend groups. Secondly a more personal experience. Even though I've mentioned about this being a USP for not only 3rd party apps, but I plan to be the plan of mine. I will not be attempting this feature at all due to (depending on how its implemented) could increase the complexity of the app greatly, time would be against me. Possibly needing internet backend, sign ups, accounts, etc. For such a small feature it is not worth the effort.

### *iCloud Backup / Sharing*

I won't focus on iCloud storage and won't be at the top of my list of priorities (even though I have mentioned it in my success criteria). This is because Apple will handle the finicky stuff such as "accounts" as obviously every user already has a iCloud account, because it's an iOS only app. Again only limiting factor is time, and making sure not to over complicate things.

### *Calendar Integration*

This is another one mentioned in my success criteria but won't be a major concern if it isn't done. This is because of the small use case, obviously you wouldn't want every event in your calendar to become a countdown. Only automatically sync some event types / calendars, etc. Larger concern maybe converting calendar events into how events are stored in the apps database records. Again only limiting factor is time, and making sure not to over complex things.

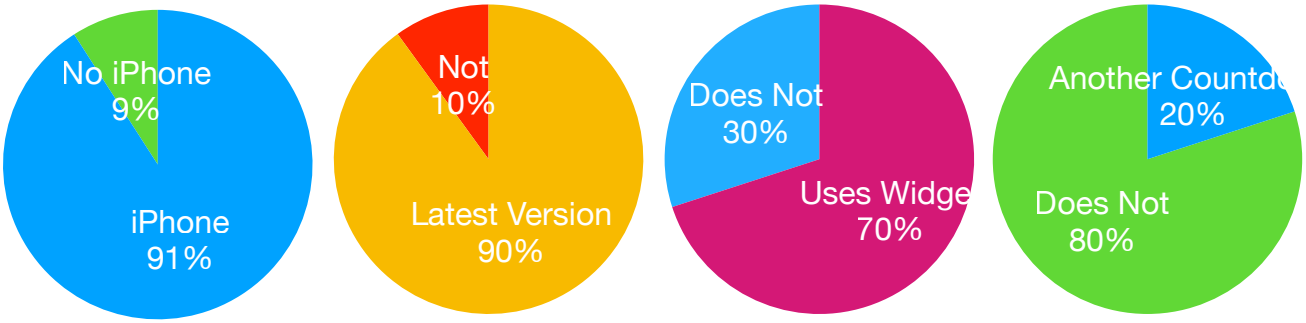
## Survey with potential users

To get a better idea for the end user of my app, I decided I would do a survey. I used the platform "Typeform" which a free service that allows you to create survey to send to people. These are the questions I included:

1. Do you have an iPhone?
2. Are you running the latest major version of iOS?
3. Do you use the new widget feature? (iOS 14)
4. Do you use any countdown apps already?
    - What app do you use? Are there any problems you see with the app?
5. Can you give any examples of events you would countdown too?
6. Any suggestions on how you could personalise a widget?
7. Any suggestions for features in my app?
8. Would you be willing to test my app for usability?

***Results***

Out the people I asked here are the responses to the questions above…




## Examples of events you would countdown too?

When asking how people what they would countdown to if they were using the app they gave the example of:

- Holidays / Christmas
- Big events (eg: concerts / prom)
- Birthdays
- Project deadlines / Exams
- Trips / Seeing someone

This shows that there is obviously events people want to countdown too. Showing a need for such widget.

## Whats wrong with your current app of choice?

For the users that already use a countdown app, I wanted to find out if they had any problems with it / missing features:
Lack of customisation was the biggest problem, which obviously with my app I plan to solve (see success criteria).

## Any suggestions on how you could personalise a widget?

I wanted to know how users themselves wanted their widgets to look like, they responded with:

- Custom background image
- Font customisation (color and face)
- Animated (eg: Fireworks)
- Change date format

Now obviously not all these requests I can take on board due to limitations of the widget itself. For example: I don't think fireworks on the widget would be possible.
Some I have already stated in my success criteria (background image, font colour). Some I haven't. Forcing a certain date format, such as only showing in months, I haven't considered. I won't append it to my success criteria, but I will take it into account.

## Any general suggestions?

Again some of these suggestions are already on my success criteria, some are not. Some I've already mentioned in limitations above as I acknowledge myself that it would be a nice feature but I won't have time to develop and write about it. Responses below:

- Notifications
- Share with friends
- Import from calendar
- Screams every time an event you're counting down to finishes
- Celebration animation

After that survey it has confirmed my analysis with the need to have the USP for customisation as that is a gap in the market at the moment.

Some I hadn’t necessarily thought of that would be quite good, like notifications.

# Design

## Overview

I want this app to focus on the home screen widget, and be the main view for the app. This is for two reasons:

- This is a new feature that would of been just released.
- As shown in the 3rd Party research section, the use of widgets in some apps is a premium feature and I wanted to make it free.

I've done a small amount personal research into how WidgetKit, the new library that Apple released back in July in relation to home screen widget, I started a few days before the release of the actual update.

## Human Interface Guidelines

When developing for iOS you have to follow strict design guidelines (Human Interface Guidelines), which yes can be irritating, but this standardisation across apps makes it easy for a user to download an app and instantly understand what everything does. Furthermore all apps follow the same design philosophy as the product they are on.

## Looking into libraries and frameworks

Before I get deep into development, thought needs to be put into how things are going to be done. This subheading will go over possible libraries and frameworks that could be used during development.

### *Database Storage Overview*

There are many different ways of doing this:

- Local database (stored on the phone)
- Online mobile backend (this will also mean account creation etc)
- As this is an iPhone app, iCloud syncing (still uses local DB)

Looking into all of these is important to make sure I choose the right one for this project. Not only for the users sake, but personal limitations such as time and experience.


***Online mobile backend***

When looking though my options two services stood out "Parse Server" and "Firebase (Google)". FireBase seems to be mainly focused on their "Realtime Database". Which is a NoSQL database with the ability to update realtime in-between platforms.

The fact that it updates realtime in-between platforms is not important as I am only developing it for iOS (Apple). Furthermore personally the use of NoSQL is not a preference I would choose due to its lack of relationships between records compared to the classic table structure of databases that I'm used to with frameworks such as MySQL and phpMyAdmin.

Whereas Parses database structure is more traditional getting close to database standards such as MySQL with a more relationship focused structure.

Personally, separating data in the logical chucks (tables) and then relating that all to a central user. Is more sensible than the "parent and child" tree approach seen in the FireBase "Realtime Database".

In this situation I could have a 'Countdown' table which contains all the data relating to a countdown, and a users table. In the 'Countdown' table the record(s) would have a field which states which user(s) said countdown is related too. This would allow a sharing feature. As multiple people could be added to one countdown record (one to many).

Parse also has the upper hand mainly due the fact I've developed with the library before on personal projects and therefore have a solid foundation on the inner workings. In addition this foundation is also solidified by its very active open source community on GitHub meaning intensive documentation not just by the main developers but the community as well. This is different to Firebase which is centralised at Google.

| | Firebase | Parse Server |
|---|---|---|
| **General Purpose** | Fast real time updates (Real time BaaS) | Open source |
| **Hosting** | Google hosting. Free up to 100 simultaneous connections . | Self hosting and Parse hosting providers. No limits. Supports Local testing and developing |
| **Custom Code** | Custom code not supported | Custom code totally supported(Cloud Code) |
| **Database** | Supports model observer scheme. Now introduced Firebase Storage to upload and download files securely | Has huge relationship based databases |
| **Push** | Support Push notifications. Firebase Remote Config to customize apps | Support Push notifications for Android, iOS. Also it is possible to send Push Notifications campaigns. |
| **Setup** | Easy setup | Quick setup on Parse Easy step by step set up guide available for migrating from Parse to Parse Server |
| **Storage** | Stores data as JSON and data backup can be uploaded to Amazon S3 bucket or Google cloud storage | No restricted time limits and No file storage restrictions. Control over backup, restore, database indexes. |
| **Provider** | Developed by Google | Developed by Facebook |
| **Ideal for** | Suitable for time applications | Suitable for building general purpose applications |

(Firebase vs. Parse Server | Back4App Blog, 2020) [3]

***Sign In With Apple***

If I use a mobile backend this will also require me to create a login system. Apart from the obvious, sign up form. I could use O-Auth systems.

As this is programmed for iPhones only this means I can make use of a 1st Part O-Auth system, called “Sign In With Apple”.

A extremely privacy conscious and simplistic system allowing users to sign up and login with one click. Authenticated biometrically with ether FaceID or TouchID.

As it is fairly new documentation can be sparse but it means login and sign up can be dealt by Apple and the Database provider of choice (which will obviously have to support such O-Auth system, both FireBase and some providers of Parse Server do).

If I did use only this method of sign up, I will not have to create a sign up / login form. All that will be needed is a clean landing page with said button, and for it to communicate with the database once login is successful.


***Local Database***

The clear option is this case is SQLite which is a lightweight database that uses the very common (and personally understood) ‘Structured Query Language’. There is very popular Swift wrapper for SQLite (“C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine”).

This would not require any accounts or 3rd party services. Meaning everything would be self contained. As said above I personally have a

lot of knowledge with SQL. Using it before on personal and business based projects and communicating with it in different languages like PHP.

The SQL file would be stored in the apps 'Documents and Data'. A connection can then be made to the database using the '*SQLite.swift*' library.

SQLite allows you to interface with the database in 2 ways:

- Raw SQL statements - SELECT id, email FROM users
- Functions built in, object like - *users.select(id, email)*

I should only need one table which will contain the countdowns, no users, less complexity. However this will make it near impossible for sharing to occur between users over the internet. However it doesn't take iCloud Syncing of the table…

***iCloud Syncing (CloudKit)***

"The CloudKit framework provides interfaces for moving data between your app and your iCloud containers. You use CloudKit to store your app's existing data in the cloud so that the user can access it on multiple devices." - *Apple Developer Website*

The highlighted parts of that are very important! "Moving Data". "Existing data". Therefore I could have a compromise between a full online database and a local one. Using the SQLite database which is already stored in 'Documents and Data' but using CloudKit to also sync it to the iCloud storage. Meaning that all user devices connected to that account will have the same countdowns. If one is made on one device it will be on the other one using 'Apple Magic' also known as the ecosystem.

Furthermore with development tools like '*Catalyst*' which make it easy to convert iPhone apps to (rough) iPad and Mac apps. Giving the app a wide possible user-base. As well as it being seamless between versions.

## Decomposing the problem

To decompose the problem, I need to break down the entire application into a series of problems. These chunks are easier to comprehend, and to tackle in terms of programming.

**These chunks are:**

- Widget system
  - Displaying the countdown
- Data manipulation & storage
  - Create database
  - Insert data into databased
  - Fetching data from the database
  - Converting dates to strings
- Add event screen
  - Collecting simple data (text, date, etc)
  - Collecting complex data (images, colours, etc)
- Main screen
  - Displaying the data to the user

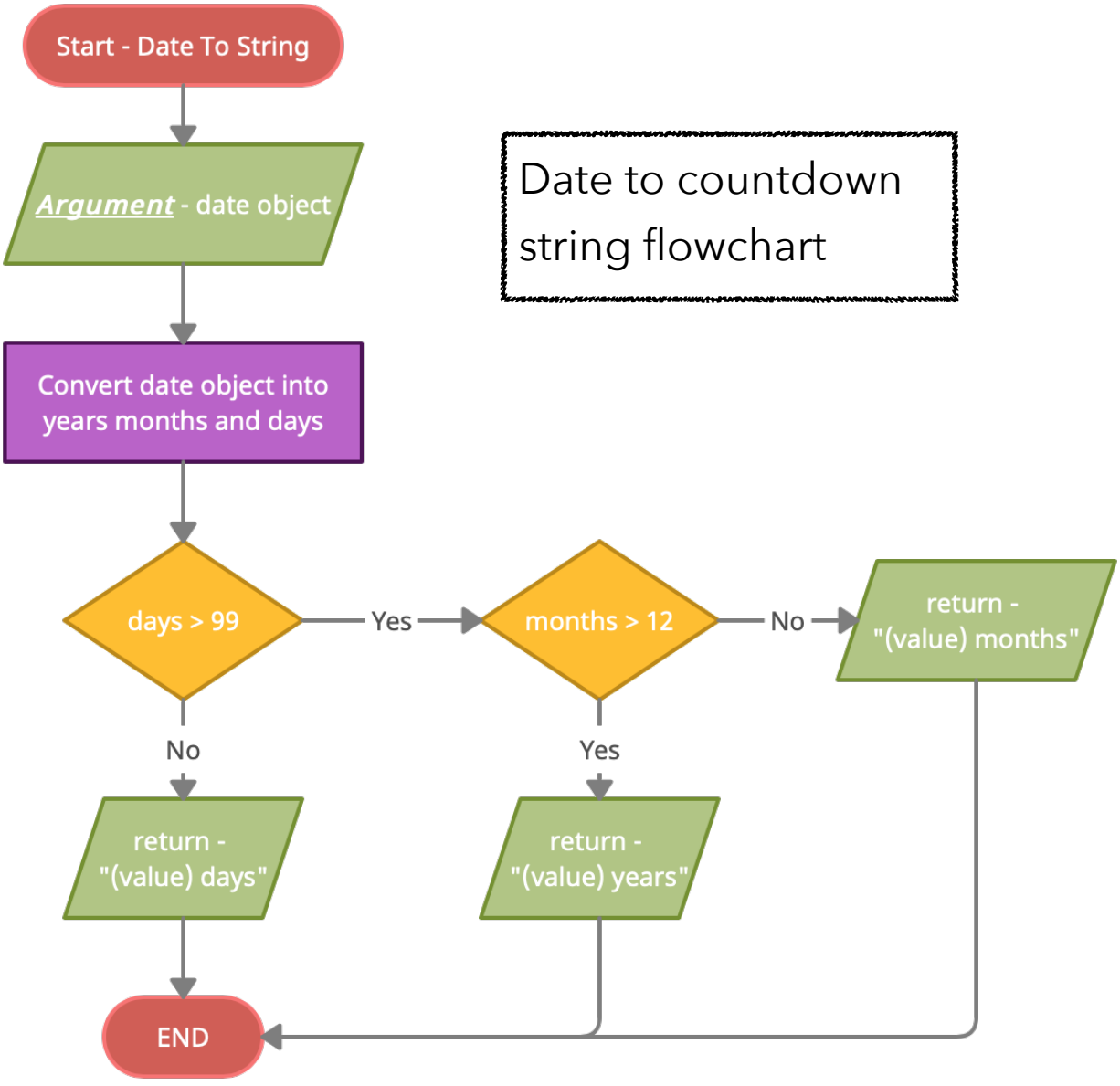
## Why they can be separated

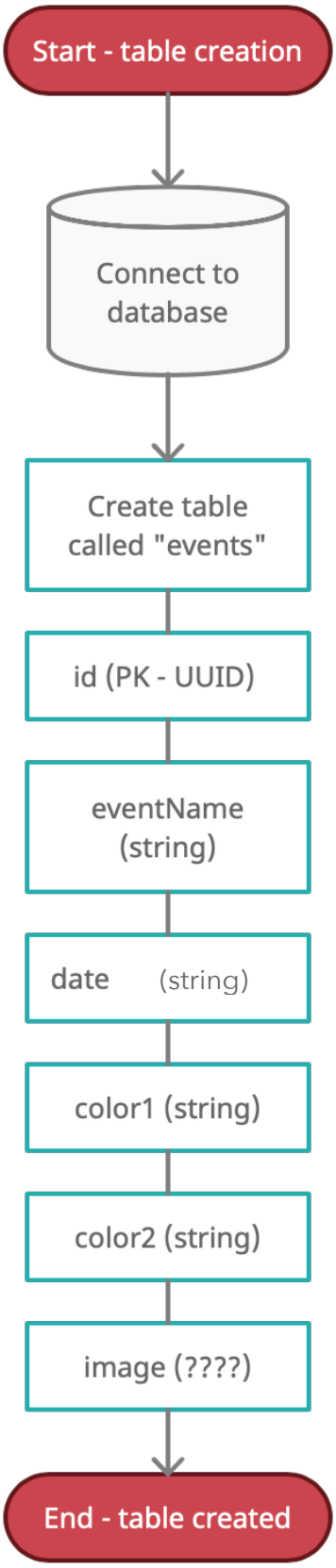
Focusing on the separation between data manipulation & storage and the rest of the project. This can easily be classed as a form of abstraction on the programming side. As I will be able to create functions and methods which do what I require: accessing the database, inserting into the database, converting dates to strings, etc. Which I can then actually call in other parts of the code for example fetching data I require before showing the main screen.

Another example is the add event screen can be separated from the rest of the project. Just like a layered stack applies in the real world (TCP/IP stack). What the other layers do is no concern as long as the output of said lawyer are relevant.

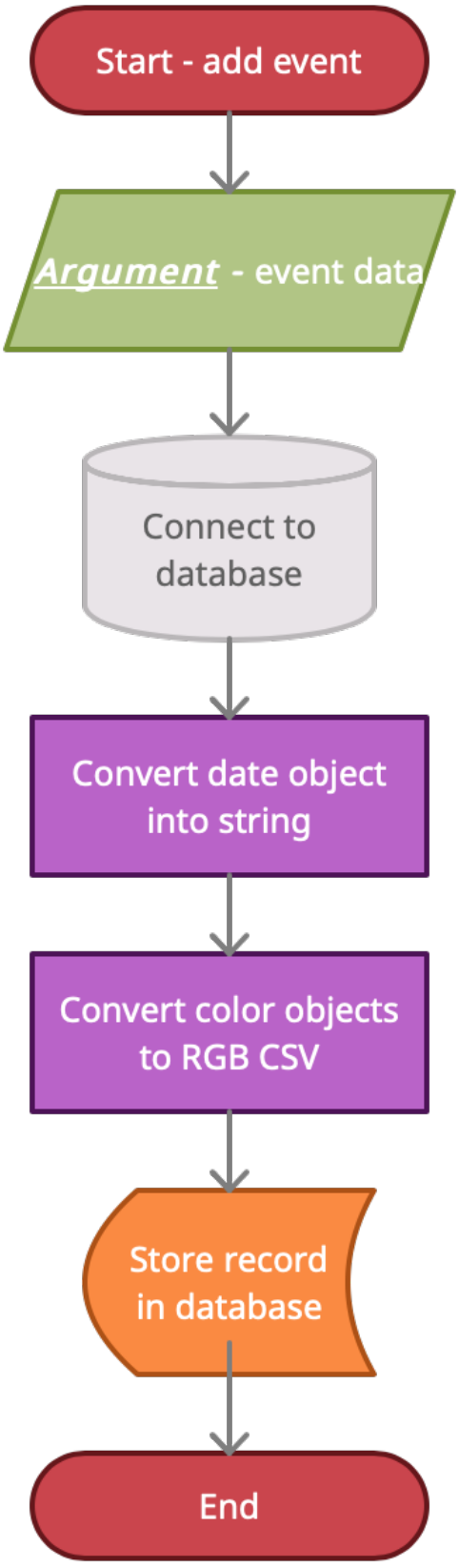
Therefore I can create and add event screen that actually collects the data, and then from there I pass the data to the next “layer” - data manipulation & storage. Which on other pages such as the main screen can go up the stack. From data manipulation & storage, to how its actually displayed by the application on the main screen.

***Algorithms Flowchart***

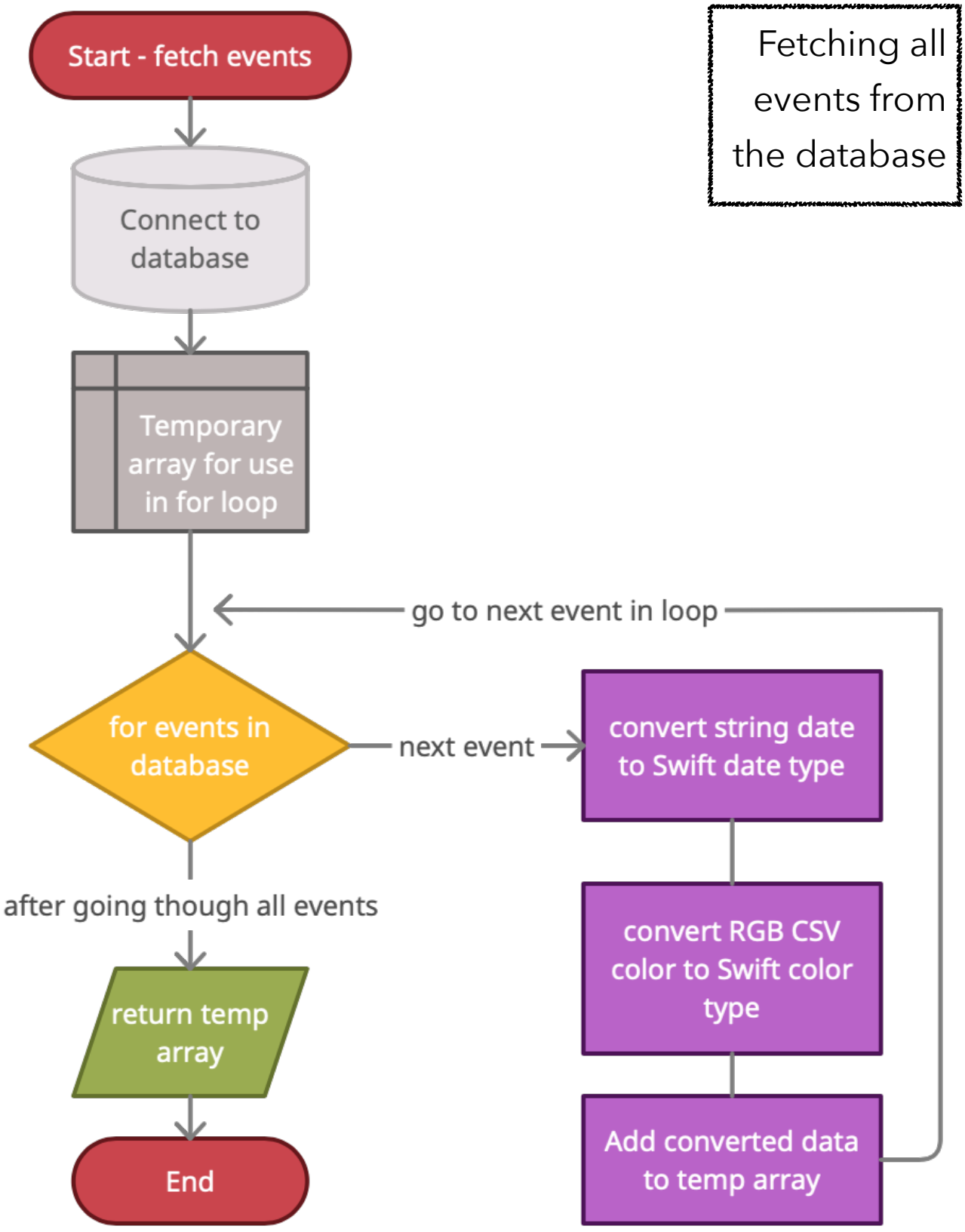





Database table structure




Adding event to the database event data


Fetching all events from the database
Start - fetch events
Connect to database
Temporary array for use in for loop
go to next event in loop
for events in database
next event
convert string date to Swift date type
convert RGB CSV color to Swift color type
Add converted data to temp array
after going though all events
return temp array
End

## Small explanation of flowcharts above

- The data conversion flowchart allows the countdown to show itself in the most relevant way, or as I call it: Dynamic. Obviously a countdown shown in years would not be relevant for a countdown days away and visa versa. So I decided that after converting the date entered into the values of: days, months, years for it to return the relevant date type depending on their values.

- Some of the flowcharts above have 'date to string' (vice versa) conversion. This will be handled by inbuilt class. Just like most modern programming languages, Swift has this sort of key thing implemented already. In this case using 'DateFormatter'.

- Converting color object to RGB (vice versa), obviously an SQLite database will not be able to store a Swift color type. Therefore I need to convert it into a data type the database will accept in this case I have chosen a string. Now at the moment I don't know exactly how I will do this. But obviously will be covered more in development. But this decision to convert to string was one I made off my own knowledge of the standard of SQLite.

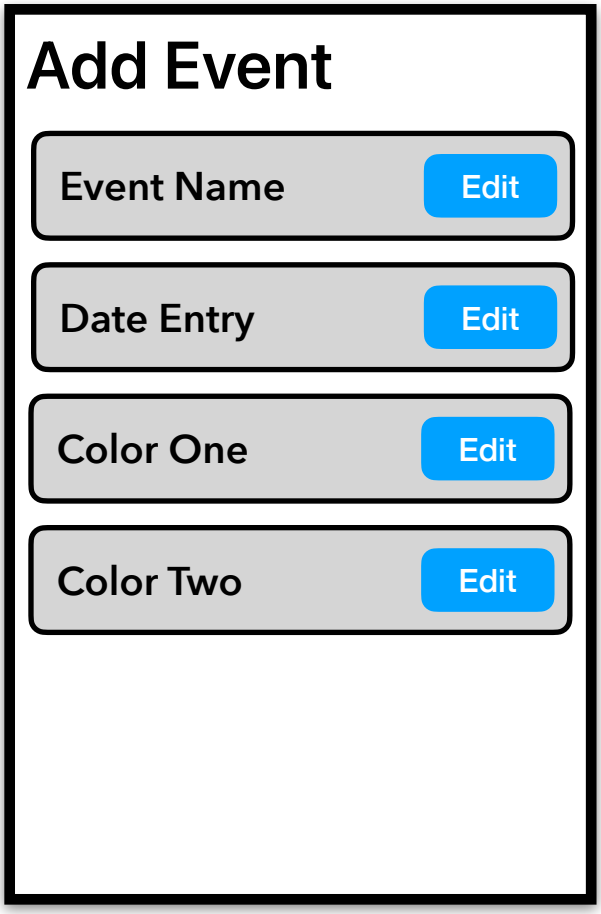
## Designing some views

***Main / Add event screen***


I like the simplicity of the card like design. With this screen showing all the events a user has created. It will make use of a scroll view incase the user has more countdowns than viewable on one screen.

However one downfall of this design is I'm not really sure at this point how I will integrate the customisable user settings into the card. Furthermore at the moment it would be a bit bland.

Accessibility wise Apple do a lot of the heavy lifting with their Human Interface Guidelines. The blue plus button in the top bar and the chevrons on the right side of the event cards, makes it clear that if you click on said view it will take you to a different screen.




The add event screen is keeping with the simplicity of the card like design and will be accessed though the plus button that was on the previous design (main screen).

I don't know exactly data will be inputted. So this design is open ended with an "Edit" button on the end. In development as I learn more about how complex date entry such as date and colour looks. This card can be adapted accordingly.

However I expect for some form of popup. So for example: colour entry will pop up a colour wheel above the view.

Again usability wise, the blue color is Apples, "blue link color" which is used for when an element takes the user to another view / action takes place.

***Widget design***


The widget design on the left is my plan for how they should look. It will be using the small widget (.systemSmall). This is because there isn't that much information to display, as you can see there is a lot of blank space around the text elements. But this is why user customisation is important, allowing them to make it their own. The background of the widget will be the gradient created by the user in the add event screen.

This view is the most important part of the app because this is what the user will see the most. In reality the user should rarely have to go into the app as once an event is created and widget is added to home screen it will just start counting down. All relevant information is there: the event in question, how many days till and the actual date it occurs, in a lighter font (as it's not as relevant, that's not what the app is about).

There is no user interaction with this view.

# Development

## General Overview of SwiftUI

I will use this as an opportunity to explain parts of SwiftUI (explained with help from reference 2) and how this helps me in the future.

### *Bit of history - UIKit and SwiftUI*

SwiftUI in terms of a programming toolkit is *VERY* new being released only in June 2019. Therefore as of date of writing. It has only recently became a year old. Before SwiftUI the toolkit that was used was UIKit. UIKit is know as a "imperative UI", this is when you the developer has to constantly update the UI when something changes. This is very messy when larger views can have many functions updating different bits of data (the "state" of the UI).

The example my reference (2) gives is perfect so to paraphrase:

"If you have a boolean property that affects the UI, that boolean has two possible states in the first place, "true" and "false". Make that two booleans and you have four states. Each would require some change to the UI. Now add strings, ints, times and dates. The complexity is endless".

### *Why SwiftUI is better and useful*

SwiftUI is declarative meaning we can tell iOS all the possible states of a view. Now this may sound complicated. But simply we can set rules on how the UI looks and then when the state changes in the app these rules are followed to get the desired state. Comparing that with UIKit where you would tell SwiftUI how to get the state and when the state itself changes. Instead this is all handled by SwiftUI.

## Starting Development

Before starting development I need to install the packages that will be required for the project. As said in the design section there were many options on how to store the data for this app. I eventually decided to take the SQLite route. The main reason is that it keeps complexity down for the time being. Allowing to focus on the main functionality of the app which is to countdown to events and then in future updates I can decide to add iCloud Syncing. As reference 4 suggests, this is very possible with a local SQL database. This shows scalability as well as maintainability in my programming as before I've fished the first version I'm already planning for future versions.

## Creating and Initialising SQLite

Prior to creating any of the UI including the main screen. Firstly I needed a database. This is because all the UI is based off this data, for example: creates a list with the events stored in the database.

### *DatabaseConnector overview*

To keep things simplistic throughout development. I decided that I should have one file with everything related to the communicating with the database. In ways abstracting away from myself. Instead rewriting the same code every time I fetched events. I had a class with methods I could call from which would just return back the data I needed which can be used for UI generation. Furthermore it also helps with maintainability because if there was errors with a certain action within the app. I could work out what function that came from and then trace back and fix the bug. Luckily with a local database there shouldn't be that many problems.

### *Creating the database*

When the app is ***FIRST LAUNCHED***, a database needs to be created with an 'events' table inside. This can be accomplished using 'UserDefaults'. As the name suggests UserDefaults is an interface which allows for permanent storage of key-value pairs, commonly used to

store user preferences inside an app. In this case on first launch of the app a Boolean UserDefault key pair - “First Launch” - is checked if it is false (or uninitialised). The database and table is created and said value is set to true. This is checked early in the app’s launch cycle inside the ‘AppDelegates’, ‘didFinishLaunchingWithOptions’ method. Which as suggested is called when the app is ready to run. It’s a method “to complete your app’s initialisation and make any final tweaks” *(Apple Docs)*. See below the code in action.

```
//didFinishLaunchingWithOptions
if UserDefaults.standard.bool(forKey: "First Launch") == false {

    //create table
    databaseConnector().createTable()

    print("First launch - db created")

    //add preset events
    //new years day
    var dateComponents = DateComponents()
    //midnight
    dateComponents.minute = 0
    dateComponents.hour = 0
    dateComponents.day = 01
    dateComponents.month = 01
    //current year + 1
    dateComponents.year = Calendar.current.component(.year,
                                        from: Date()).advanced(by: 1)

    //add preset event to database
    databaseConnector().addToDB(data: eventObject(name: "New Years Day", date:
    Calendar.current.date(from: dateComponents)!, colors: [.red, .orange]))

    //set “First Launch” to true
    UserDefaults.standard.set(true, forKey: "First Launch")
} else {
    print("not first launch")
}
```

Therefore, when the app is first launched, this code is executed. First running the function ‘createTable’ in the class ‘databaseConnector’. To actually create the database and ‘events’ table (code is shown below). After that it populates the database with one record so its not completely empty when the user is taken to the main screen. In this case New Years Day for the next year - ‘advance(by: 1)’  - from when the app is first opened.

Adding events will be shown later, this written more in a way that makes sense rather than perfect chronological order.

```
public class databaseConnector {

    //global vars most functions need
    let eventsTable = Table("events")

    //generic structures associated with a type
    let id = Expression<Int64>("id")
    let eventName = Expression<String>("name")
    let date = Expression<String>("date")
    let color1 = Expression<String>("color1")
    let color2 = Expression<String>("color2")
    let bgIMG = Expression<String>("bgIMG")

    //where the databse is stored in 'Documents and Data'
    let path = NSSearchPathForDirectoriesInDomains(
                .documentDirectory, .userDomainMask, true).first!

    func createTable() -> Bool {
        do {
            //creating the database file
            let db = try Connection("\(path)/events.sqlite3")

            // create table
            try db.run(eventsTable.create { t in
                    t.column(id, primaryKey: true)
                    t.column(eventName)
                    t.column(date)
                    t.column(color1)
                    t.column(color2)
                    t.column(bgIMG)
            })

            //table make successfully
            return true
        } catch {
            //idk what went wrong
            return false
        }
    }
...
```

***Standardisation of event data***

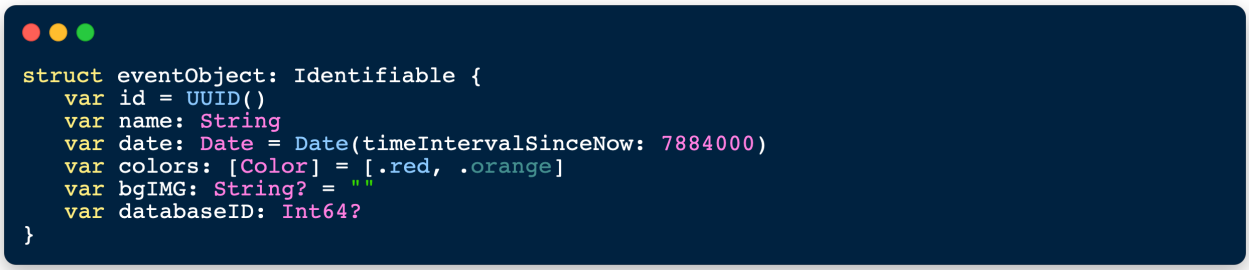
To make sure that event data is effectively stored as well as type-safe. I have used a struct. Structs are a composite data structure which can store many different properties with different types under the same label.

Compared to using arrays which are indexed based, this would be very hard to maintain, [3] means nothing in two months after not looking at your code. However 'eventObject.date' is very clear what that is.

I can also make this struct conform to the protocol ‘Identifiable’ which means I will be able to use it ease in a SwiftUI list. This will mean that a SwiftUI list can take an array of these objects and loop though them. Telling the difference between each ‘eventObject’.

Furthermore with the use of optionals (question mark after data type) this means I can guarantee properties such as event name and date. Which out of anything are the most important values, anything else, default values can be used. This should prevent crashes from ‘optional unwrapping’.

This standardisation means throughout the program (especially outside of ‘databaseConnector’) any data that references an event will be in this format that’s easy to use.



```
struct eventObject: Identifiable {
    var id = UUID()
    var name: String
    var date: Date = Date(timeIntervalSinceNow: 7884000)
    var colors: [Color] = [.red, .orange]
    var bgIMG: String? = ""
    var databaseID: Int64?
}
```

### *Create event function*

As stated above when the database and table are created on first launch an event is also added. This is so there wouldn't be an empty list when the main screen is eventually showed. Therefore a way to add events is needed. As well as obviously in the future when users add them themselves

```
public class databaseConnector {
   func addToDB(data: eventObject) -> Bool {
      do {
         let db = try Connection("\(path)/events.sqlite3")

         //convert swift date object to string
         let dateFormate = DateFormatter()
         dateFormate.dateFormat = "yyyy-MM-dd"
         let stringDate = dateFormate.string(from: data.date)

         //convert swiftUI color to string color array
         let colorString1 = "\(data.colors[0].components.red),\
                  (data.colors[0].components.green),\(data.colors[0].components.blue)"

         let colorString2 = "\(data.colors[1].components.red),\
                  (data.colors[1].components.green),\(data.colors[1].components.blue)"

         //add said data into table
         try db.run(eventsTable.insert(eventName <- data.name, date <- stringDate,
                                       color1 <- colorString1, color2 <- colorString2,
                                       bgIMG <- data.bgIMG ?? ""))

         return true
      } catch {
         return false
      }
   }
...
```

As you can see in the code above this 'addToDB' function take in the parameter, data. This argument has to be the composite data structure 'eventObject'. This can then be used inside the function manipulating the data to a form I can input in the database. In this case converting data objects into a text format. As well as colour objects into CSV RGB strings. It can then be inserted into the database. I did look into if I could store the values as their Data/Colour objects. But this method of converting back and forth when inserting and fetching was the best solution.

Additionally as said above, the 'databaseConnector' class is an abstraction layer for me. So once it's done it can just be used, calling the functions. So this converting isn't really an issue.

### *Fetching Events*

Leading on from creating events, I need a way to fetch events back from the database. Again this is method inside 'databaseConnector'.

```
public class databaseConnector {
   func fetchEvents() -> [eventObject]? {
      do {
         let db = try Connection("\(path)/events.sqlite3")
         var tempArray = [eventObject]()

         for event in try db.prepare(eventsTable.order(date)) {
            //convert string date to Date object
            let dateFormate = DateFormatter()
            dateFormate.dateFormat = "yyyy-MM-dd"
            let dateObject = dateFormate.date(from: "\(event[date])")

            //convert string to color
            let colorOneArray = event[color1].components(separatedBy: ",")

            let colorOne = Color(red: (colorOneArray[0] as NSString).doubleValue,
                                 green: (colorOneArray[1] as NSString).doubleValue,
                                 blue: (colorOneArray[2] as NSString).doubleValue)

            let colorTwoArray = event[color2].components(separatedBy: ",")

            let colorTwo = Color(red: (colorTwoArray[0] as NSString).doubleValue,
                                 green: (colorTwoArray[1] as NSString).doubleValue,
                                 blue: (colorTwoArray[2] as NSString).doubleValue)

            //add to temp array and return
            tempArray.append(eventObject(name: "\(event[eventName])",
                                         date: dateObject!,
                                         colors: [colorOne, colorTwo],
                                         bgIMG: event[bgIMG], databaseID: event[id]))
         }

         return tempArray

      } catch {
         print(error)
         return nil
      }
   }
...
```

This creates a temporary array, then starts to loop through all the events in the list (ordered by upcoming event - date). As mentioned above, in the create event function, the SQL database couldn't store the Swift Date/Colour objects. Therefore that data needed to be stored in a string format in the database. Converting to text on the way in and back into Swift object form when fetched out.

You can see this in the code snippet above:

- Converting string date in database into Date() object.
- Taking the CSV formatted string in the database, converting that into an array of numbers.
- Using that newly created array of numbers (RGB respectively), and creating colour objects.

### *Testing database and its functions*

Now the basic functionality of the database is done, even without UI I should be able to use the console to see the database being created on first boot, as well as the 'New Years Day' event. After that all events in database should be printed (one event in this case).

```
<Console - Countdown Widget - First Boot>
First launch db create: true
Added NYE: true

[Countdown_Widget.eventObject(id: 2F7ED6C6-0D73-450D-A8EF-0C77C17525A5, name: "New Years
Day", date: 2022-01-01 00:00:00 +0000, colors: [#FF3B30FF, #118AF19FF], bgIMG:
Optional(""), databaseID: Optional(1))]
```

```
<Console - Countdown Widget - Second Boot>
not first launch

[Countdown_Widget.eventObject(id: 2F7ED6C6-0D73-450D-A8EF-0C77C17525A5, name: "New Years
Day", date: 2022-01-01 00:00:00 +0000, colors: [#FF3B30FF, #118AF19FF], bgIMG:
Optional(""), databaseID: Optional(1))]
```

As you can see, on first boot everything was created successfully. But on the second boot nothing was needed to be created. Then just as expected the event that was created was there, and it was printed to console.

Now I know these functions work correctly I can start creating basic UI using these functions.

## Main Screen Development

Now there is a database with some sample data, I can start to create the user interface. In SwiftUI subview (views inside views) are used. Creating elements as their own views are recommended by Apple themselves. Due to SwiftUI's nature extracting one large view into many subviews combined doesn't have any effect on the loading speed of the view. So it is recommended to allow maintainability as well as just general clean code and good practice. Firstly I need to create a 'card' view to show a countdown which I can reuse in a list view.

```
struct CardView: View {

    //system dark / light mode?
    @Environment(\.colorScheme) var colorScheme

    //pass in event data
    @State var event: eventObject

    @State var stringDate: String = ""
    @State var bubblePreview: [String: String] = ["": ""]

    var body: some View {
        ZStack(alignment: .leading) {
            //Card (rounded rectangle)
            RoundedRectangle(cornerRadius: 8)
                .frame(height: 100, alignment: .center)
                //change background colour depending on light / dark mode
                .foregroundColor(.init(.sRGB, white:
                                       (colorScheme == .dark) ? 0.12: 0.92, opacity: 1))

            HStack(alignment: .center) {

                //views ontop of views - Z axis
                ZStack {
                   //circle on the left with date in
                   Circle()
                      //fill circle with gradient using colour user chose
                      .fill(LinearGradient(gradient: Gradient(colors: [event.colors[0],
                                           event.colors[1]]), startPoint: .topLeading,
                                           endPoint: .bottomTrailing))
                    VStack {
                        //data ontop of circle
                        //how many days, weeks, months, etc.
                        Text(bubblePreview["value"] ?? "--")
                            .font(.largeTitle)
                            .bold()
                            .foregroundColor(.white)

                        Text(bubblePreview["type"] ?? "-----")
                            .font(.system(size: 14))
                            .foregroundColor(.white)
                            .italic()
                            .offset(x: 0, y: -3)
                    }
                }
                .frame(width: 85, height: 85, alignment: .center)
                .shadow(radius: 3)
                .padding(8)

                VStack(alignment: .leading) {
                    //name of the countdown
                    Text(event.name)
                        .font(.headline)
                        .fontWeight(.bold)
                        .lineLimit(2)
                        .foregroundColor(.primary)

                    //When countdown finishes (eg: February 15th 2020)
                    Text(stringDate)
                        .font(.subheadline)
                        .fontWeight(.light)
                        .foregroundColor(.secondary)

                }
            }
        }
    }
}
```

### *Card View (Subview)*

The code on the left is a minimised version of the card view code. I will go into detail about certain sections.

But there is the basic UI elements of the card. Displaying the date, name of countdown and how many days till, in a fancy personalised fashion.

None of the card view actually contains any calculations. Again that is abstracted away. I have created methods for that.

### *How design works in SwiftUI*

To stop me from repeating myself in every section, I am going to explain how designing views work in SwiftUI. A lot of the views are going to look the same: rounded rectangle, icons, text, etc.

SwiftUI uses view modifiers, to customise inbuilt views (eg: 'Text', 'RoundedRectangle', 'Image'). You can see these above as they are below to view itself and indented, with a period at the beginning of the modifier - "***.foregroundColor***". Below is a table which explains common modifiers I use throughout this app. This can be referenced to later if any confusion about the views design.

| Modifier | Paramaters | What I use it for? |
|---|---|---|
| ***.frame*** | • Height - How high the view should<br>• Width - How wide the view should be.<br>Only one can be defined if required<br>• Alignment - how that view is aligned if it can't be the size defined above. | Defines the dimensions of said view on the screen. |
| ***.padding*** | • Array of edges - Which edges should the padding be applied too, for example: [.leading, . top]<br>• Length - the size of the padding<br>If no value provided the default value is used. The default padding amount / padding is applied to all edges. | Spacing between views, as well as keeping UI elements from going edge to edge (like if they were in a container). |
| ***.font***<br>***.bold***<br>***.italic***<br>***.fontwieght*** | Some take parameters, to do more granular control. Some modifiers do general things, for example '.bold'. | Used to style text, give some hierarchy to the design with different weights and sizes. |
| ***.navigationBarTitle*** | • Title - String | Adds a large title to navigation bar, will see affect of this in future screenshots |
| ***.forgroundColor*** | • Colour to use - what colour should the view be set to. | Set font colour,<br>Colour of shapes, etc |
| ***.fill*** | • View to use - the view that will be used as a background for the view its applied to. | The 'circle view' to add the gradient. As the gradient is a view. |

### *DateConverter*

As said above the card view, is not actually doing any calculations. Just taking in data and displaying it using SwiftUI. The main calculation is abstracted away in a method called 'DateToString'. The method takes in the date object as a parameter, using that to work out how many: days, months or years till that date. By abstracting away this code from the 'card' makes the app more maintainable, as well as being able to reuse this code in the future; keeping the UI and calculations separate.

```
class DateConverter {
    func DateToString(dateObject: Date) -> [String: String] {

        //Get current date (from midnight) workout how long to countdown
        let startOfDay = Calendar(identifier: .gregorian).startOfDay(for: Date())
        let toDays = Calendar.current.dateComponents([.day], from: startOfDay, to:
                     dateObject)
        let toMonth = Calendar.current.dateComponents([.month], from: startOfDay, to:
                     dateObject)
        let toYear = Calendar.current.dateComponents([.year], from: startOfDay, to:
                     dateObject)

        //CREATE READABLE STRINGS FROM SAID VALUES

        var returnResult = [String:String]()

        //only allow 2 digits therfore max 99 days
        if toDays.day! > 99 {
            //if more than 99 then show month / years
            if toMonth.month! > 12 {
                returnResult["value"] = String(toYear.year!)
                returnResult["type"] = "Years"
            } else {
                returnResult["value"] = String(toMonth.month!)
                returnResult["type"] = "Months"
            }
        } else {
            returnResult["value"] = String(toDays.day!)
            returnResult["type"] = "Days"
        }

        //if event past / current day
        if (toDays.day! < 1) {
            if toDays.day! < 0 {
                returnResult["value"] = "Clock Emoji"
                returnResult["type"] = "Past"
            } else {
                returnResult["value"] = "Party Emoji"
                returnResult["type"] = "Now"
            }
        }

        //removes 's' (plural) if value is equal to 1
        if (Int(returnResult["value"]!) == 1 && (returnResult["value"]! != "Past" ||
                                                 returnResult["value"]! != "Now")) {
            returnResult["type"]!.removeLast()
        }

        return returnResult

    }
}
```

This method takes in the date object of the event, and does all the calculations to work out everything. It then creates human readable string. In a ‘Key : Value’ style array. 'Value' being the how many of date type (eg: Days) till the event. Then ‘Type’ being the human string of said date type (eg: the string “Months”).

The last bit of code to be done is to remove any plural ‘s’ if the value is equal to one. So it doesn’t sound strange (eg: “1 Months”).

### *Card View (cont.)*

Now there is a method which will do the majority of execution. I can use SwiftUI’s ‘onAppear()’ to preform actions when the view is loaded in.

```
.onAppear() {
    let toString = DateFormatter()
    toString.dateStyle = .long
    stringDate = toString.string(from: event.date)

    bubblePreview = DateConverter().DateToString(dateObject: event.date)
}
```

In this case when the view is loaded in, the event data is used to to create the ‘stringDate’ (“15 February 2020”). ‘DateToString’ method returns a value used to create the actual countdown text (“10 days”).

### *Creating the list*

Now that I’ve created a card view as well as the database connector, I can actually start creating the main UI. Firstly I need the main view, ‘ContentView’, to have all the data. As ‘ContentView’ is root view, I can fetch the data when it's being called. So that means before it’s even shown to the user the data has already been fetched.

```
var body: some Scene {
    WindowGroup {
        ContentView(events: databaseConnector().fetchEvents() ?? [eventObject]())
    }
}
```

```
struct ContentView: View {

    @State var events: [eventObject]
...
```

In the two code snippets above you can see, 'ContentView', the root view being called in the 'entry point' of the apps lifecycle. With an argument inside being the function that fetches events. This means the returned value will be passed into the variable events in 'ContentView'. The reason why events is a parameter of 'ContentView' is because I didn't initialise the variable, only set its datatype (after colon - "[eventObject]" - array of event structs). Consequently there needed a value before 'ContentView' could be called.


The event data is now available in 'ContentView' this can be used to create the list, not just too print to console. As said above the 'eventObject' struct is Identifiable therefore each object is unique. This can be done by making one of the properties a 'UUID()', this is a universally unique value that can be used to identify types, interfaces, and other items.

```
struct eventObject: Identifiable {
    var id = UUID()
    var name: String
    var date: Date = Date(timeIntervalSinceNow: 7884000)
    var colors: [Color] = [.red, .orange]
    var bgIMG: String? = ""
    var databaseID: Int64?
}
```

To make a list now is simple…

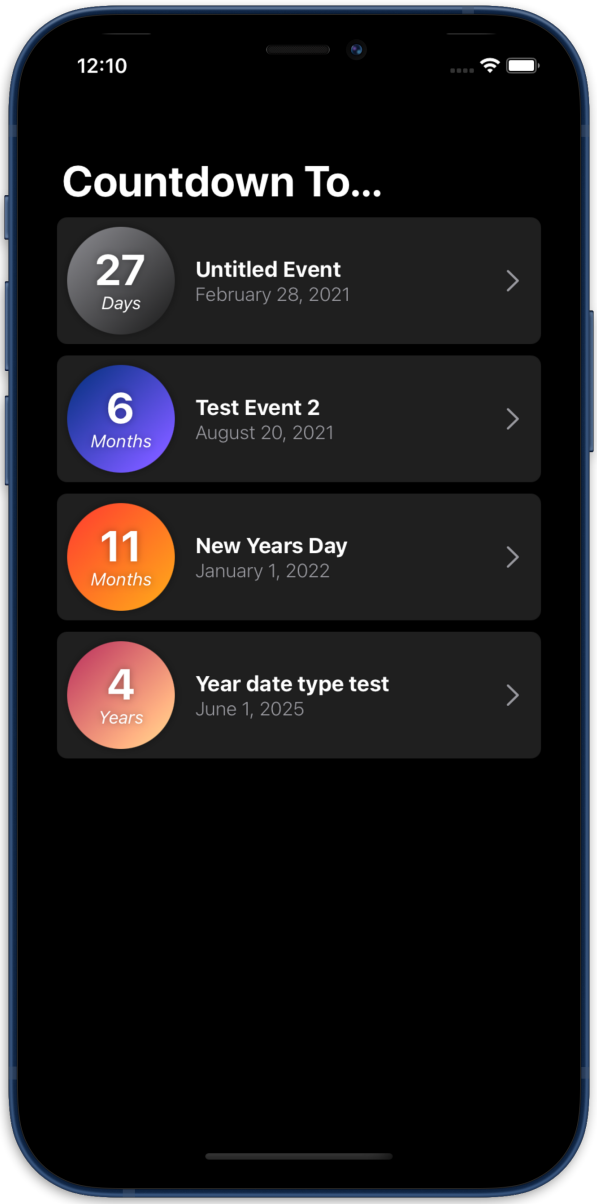
```
ForEach(events) { event in
    CardView(event: event)
        .padding([.leading, .trailing])
}
```

Looping through all the events fetched from the database and for each one creating a 'CardView' with that events data. Easily making a list.

To test to make sure the loop is working correctly I need to manually add data to the list before the program starts and see if the events are shown.

### *Testing basic main screen*

Just before the events are fetched before ‘ContentView’ is called I used the method ‘adToDB’ to create some fake events. When I started my app, all the events I added, plus 'New Years Day' which was already there, were in a list.




The basic main screen is now complete. At this point the app would be able to countdown to any date that were in its database.

I could do some testing now by manually inputing erroneous dates such as last week. However this would not be a fair test as all validation would be done when the user is adding the events to the database. So if this caused an error now that would be irrelevant.

### *Relating back to the success criteria*

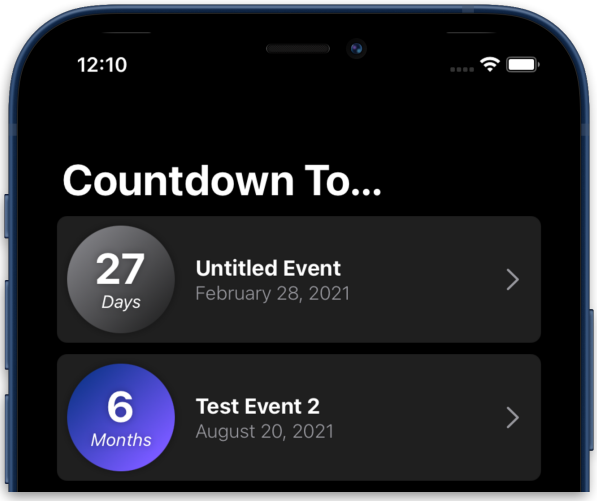
As one of the main functions of the app will be the ‘Widget’ system, everything up to now is just building up to that, so the only thing that can be half ticked off would be: “Updates daily, as countdown decreases”. Even though that isn’t the widget, these countdowns in app would decrease as it gets closer to the date.

## Event Creation Screen

This screen will simply take in all the input required for event creation which then can be entered into the database.

### *NavigationView*

This is key feature of SwiftUI, allowing to push and pop views from the screen giving the app a hierarchical structure. As well as giving views titles and buttons to the top of the screen. I have already done one of these things already. Which is add a title to the main screen.




```swift
//sets title for view as seen above
.navigationTitle("Countdown To...")
.navigationBarItems(leading: EditButton(), //built in system button with features
                    trailing: HStack {
                      //when button clicked navigate to empty page with “hello world”
                      NavigationLink(destination: Text("Hello World") {
                         //system icon - built in
                         Image(systemName: "calendar.badge.plus")
                            .font(.title3)
                      }
                      .padding(.trailing, 6)
                    })
```

The code for adding a title to view as well as adding buttons to the navigation bar. At the moment the only button added to the bar was  a ‘Create Event’ buttons. However at the moment, this button only redirect to an empty view with the text “Hello World” in it. As shown in the code snipped above - navigation link.

"A NavigationLink defines a destination for a navigation-based interface and allows the user to perform that navigation” - Apple Documentation. Navigation takes in one parameter, ‘destination’, this is the view that will be displayed as

child to the current view (AKA: a detail view). The body of this struct contains how the view should look. In this case a system icon of a calendar.

As shown in the screenshot, as well as adding a button in the navigation bar, I have also created a button which appears at the end of list. Which if also clicked will take you to a detail view. In the future I will just be able to call the struct 'Create Event' view. Making the buttons go direct to that screen.

From now on, mention of design choices will be limited, explaining each views design and how it was could would be repetitive and make this document longer than "War & Peace".

***The screen itself***

This screen will show the clear advantages of splitting off a screen into separate views in their own structs. As said above, this is an Apple recommend practice and doesn't slow down loading of the screens. Personally it also makes the program more maintainable as you aren't looking though spaghetti code, you can go directly to the section that's causing the problem.

***Sections (Inputs) Required:***

- Event date input
- Event name
- Two colours that creates a gradient
- Import from camera roll, background image for widget

Firstly I need to create a parent view of sorts to hold all these child views - 'addEvents'. Inside this there will be a a vertical stack of views (VStack), creating the overall layout.

***Adapting the design***

I've decided to adapt the design that I originally made, instead of just having a list of inputs, I took the circle from the main page and blew it up. Allowing the user to see in real time the changes they are making.

This gives the user more feedback. This will be vital for the gradient, instead of just selecting colours and seeing what it looks like after you complete creating the event.

This will also use '@State' to its full potential. When a state changes, the whole view is reloaded and subviews inside of it are reloaded. Therefore when the variable one colour changes, the whole view will reload and a new gradient will be shown on the 'real time circle'.

```
struct addEvent: View {

    //data the user picks
    @State private var datePicked = Date()
    @State private var color1 = Color.gray
    @State private var color2 = Color.primary
    @State private var eventTitle = ""
    //initialize state vars with default values

...
```

However for now, I'll focus on actual data input. Then later using these variables in the future for the 'real time circle'.

***Date Picker***

Firstly, I'm going to discuss what a @Binding is and how this relates to the @State above, that are declared parent view of 'addEvent'. A binding creates a two-way connection between a @State and its @Binding. As the name suggests, it binds the variable created in the parent view - which is the so called 'source of truth' - to the property in the child view. It can read and write. Therefore changing the binding will change the @State, as said reloads the view with this new data.

```
struct addEvent: View {

    @State private var datePicked = Date()

    var body: some View {
        VStack(alignment: .center) {
            DatePickerCell(datePicked: $datePicked)
            //calling the data picker view and passing in the binding of that state
        }
    }
}


struct DatePickerCell: View {

    @Binding var datePicked: Date

    ...
}
```

For all the data inputs (as well as elsewhere on this app) I have standardised how things should look. In the case of the inputs on this screen. A grey rounded rectangle (shade depends on dark / light mode), which has text on the left for what that is there for, and on the right an icon or the way you actually interact with the input.


Whats shown above is how the design and workings of that button are hand in hand. The rounded rectangle, the text and then the actual date picker itself which as shown in the code above the selected date is then written to the variable 'datePicked', as well as the fact it will only allow selected dates from 'Date()' (the default date object, current date) onward, shown by the ellipse. This is a form of ***validation*** by restricting the user input. It is impossible to select a date that is before the current date. This is because this app will not have a 'days since'. Countdown only.

### *Colour Picker*

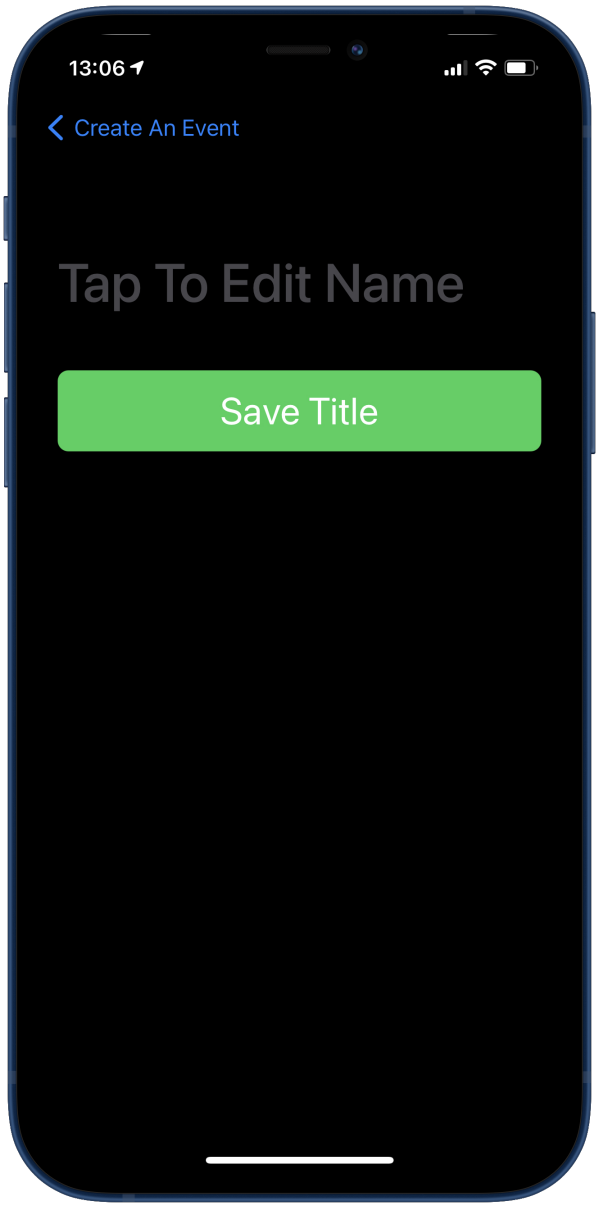
In this view, it will contain two inputs of ‘colour pickers’ these can then be used to create a gradient with respective colours.

```
struct ColorPickerCell: View {

    @Binding var color1: Color
    @Binding var color2: Color

    //two boxes here becuase they deal with the same thing, colour
    var body: some View {
        ZStack {
            RoundedRectangle(cornerRadius: 8) //grey box
                .foregroundColor(Color(.systemGray5))
                .frame(height: 58, alignment: .center)

            HStack {
                Text("Primary Colour") //text describing
                    .font(.title3)
                    .foregroundColor(.primary)
                    .fontWeight(.semibold)
                    .padding(.leading, 10)

                //SwiftUI Color picker, selected colour writen to the varable ‘colour1’
                ColorPicker("", selection: $color1, supportsOpacity: false)
                    .padding(.trailing, 10)
            }
        }
        .padding([.trailing, .leading])

        //second colour picker
        ZStack {
            RoundedRectangle(cornerRadius: 8) //grey box
                .foregroundColor(Color(.systemGray5))
                .frame(height: 58, alignment: .center)

            HStack {
                Text("Second Colour") //text describing
                    .font(.title3)
                    .foregroundColor(.primary)
                    .fontWeight(.semibold)
                    .padding(.leading, 10)

                //SwiftUI Color picker, selected colour writen to the varable ‘colour2’
                ColorPicker("", selection: $color2, supportsOpacity: false)
                    .padding(.trailing, 10)
            }
        }
        .padding([.trailing, .leading])
    }
}
```


Everything I’ve said above applies now, validation, saving to the binding variable, etc.

***Event Title***

At first I tried to implement the text editing into the actual button, meaning that when you changed the event tittle, the text on the button would change from “Event Name” to whatever you typed. This is my eyes would of been more seamless. However there were a few problems those being:




- When the keyboard came up, the UI freaked out too much and sometimes covered the button itself depending on device size.
- Longer event titles would run over.
- It couldn’t make the ‘text field’ activate when a button was clicked as that capability wasn’t directly possible in SwiftUI, therefore they would have to click the text which isn’t consistent with the rest of the design.

Therefore I decided to stray from this main design and have a button that takes you to a child view which would mean the user would click an ‘Edit’ button which would take you to said child view where then can then enter the event title in a text field. This solves the problems above, however I personally think its not a  seamless design because you have to go to a separate screen. This is an example of compromising design for overall user experience and usability (UX).

```
struct TitleEvent: View {

    @Binding var text: String

    var body: some View {
        ...
        // When whats inside navigation link is clicked go to 'EditText' view
        NavigationLink(destination: EditText(text: $text)) {
            ... // DESIGN STUFF LIKE BEFORE
        }
        ...
    }
}

struct EditText: View { // Seperate child view to enter text

    @Environment(\.presentationMode) var presentationMode: Binding<PresentationMode>
    @Binding var text: String
    // Another @Binding, data will propogate to source of truth at root parent - @State

    var body: some View {
        VStack {
            TextField("Tap To Edit Name", text: $text) // Text entry, saves to var text
                .font(.system(size: 40, weight: .medium, design: .default))
                .padding()
                .accentColor(.primary)


            Button(action: {
                // when button clicked navigate back to parent page - 'editEvent'
                self.presentationMode.wrappedValue.dismiss()
            }) {
                ZStack {
                    RoundedRectangle(cornerRadius: 8) // green save button
                        .foregroundColor(.green)
                        .frame(height: 60, alignment: .center)

                    HStack {
                        Text("Save Title") // text on button
                            .foregroundColor(.primary)
                            .font(.title)
                    }
                }
                .padding()
            }

            Spacer()
        }
    }
}
```

As you can see in the code above, I have two views. One being 'TitleEvent'. Which is the actual button on the 'editEvent' page and the other - 'EditText' - being the child view which is navigated to when the button is clicked. This is done in the same way as done before to navigate from main page to edit page, 'NavigationLink', as seen above.

Both views have a @Binding variable for text (where the event title is stored). The @Binding in 'TitleEvent' is passed into the @Binding for 'EditText'. When the user inputs text in 'EditText', the variable will also change in 'TitleEvent' and therefore, in the source of truth @State variable in 'addEvent' view (data propagates up).

This variable (text) is changed automatically live, I do not have to call any update function or 'press enter' for the string to be saved. As the user types it constantly updates this variable. The save button actually doesn't save anything. All it does is dismiss the view (navigate back to previous view). Data will be added to the database later.

### *Putting it all together*

Now I have the main input views sorted. I can add them all to the main view, combining them together. As well as this, I need to add a button to the bottom of the view which would create the event and add it to the database.

```
struct addEvent: View {

    @Environment(\.presentationMode) var presentationMode: Binding<PresentationMode>

    //Data the user picks
    @State private var datePicked = Date()
    @State private var color1 = Color.gray
    @State private var color2 = Color.primary
    @State private var eventTitle = ""

    var body: some View {
        // Vertically stack all input views
        VStack(alignment: .center) {
            DatePickerCell(datePicked: $datePicked)

            TitleEvent(text: $eventTitle)

            ColorPickerCell(color1: $color1, color2: $color2)
        }

        Spacer()

        // Button to add event to database
        Button(action: {
            // If no title, set it to a preset value
            if eventTitle == "" {
                eventTitle = "Untitled Event"
            }

            // Add event to database using user inputed data (premade method)
            databaseConnector().addToDB(data: eventObject(name: eventTitle,
                                        date: datePicked,
                                        colors: [color1, color2]))

            // Dissmiss view, go back to main view
            self.presentationMode.wrappedValue.dismiss()
            print("event added") // Print to console to show something has happend
        }) {
            // The design of the button
            ZStack {
                RoundedRectangle(cornerRadius: 8)
                    .foregroundColor(editMode ? .yellow: .green)
                    .frame(height: 60, alignment: .center)

                HStack {
                    Text(editMode ? "Save Changes": "Add Event")
                        .foregroundColor(.white)
                        .font(.title)
                }
            }
            .padding()
        }
    }
    .navigationBarTitle((eventTitle == "") ? "Create An Event": eventTitle)
    // If no event title: title of view is 'Create An Event'
}
```

As seen on the previous page, the views are combined in the vertical stack (VStack) and a button is added below. Which when clicked firstly does some basic last minute validation, for example: if event title is empty set title to 'Untitled Event'. After that the data is added to the database and the user is taken back to the main screen because the view is dismissed. Lastly some text is printed to the console to show that all of the execution has finished.

## Add Event Testing

Now I've built the basic foundations for adding an event, I need to do some testing so I can improve the system if any problems are found.

### *Date Picker*

As said above, as the date picker is already built into SwiftUI. It's going to work as required. Furthermore it has built in validation. Which in this case, forced the date picker to not allow any dates before the current date.


### *Text Input*

Text input obviously means the user can type in whatever they want into the field, therefore I should test a few different string to see how it reacts.

### *Colour Inputs*

Again the 'Color Picker' views are built into SwiftUI so there shouldn't be any problem with the user input side, this is more testing my conversion code shown in the 'databaseConnector' section as said the database can't store SwiftUI 'Color' objects so it had to be converted CSV RGB.

What I noticed first was the fact that when the 'add Event' page was dismissed the events didn't show on the main screen till the app was reloaded. I will fix that later. For now I'll just reboot to test if it's working.

## *How I will display test data*

| Testing / Justification | Test Data | Expected Result | | Actual Result |
|---|---|---|---|---|
| What I'm actually testing, why it needs to be tested. | Test data 1 | What I expect from that data | | Did what I expect actually happen, or did something different, do I need to change anything because of it.. |
| | Test data 2 | ^^ | | ^^ |
| | Test data 3 | ^^ | | ^^ |

## *Colour coding (expected result):*

**Normal** - Expected normal user input

**Borderline** - Accepted by the system, could possibly causes issues.

**Erroneous** - Rejected by the system, not allowed

## *Testing Date Picker*

| Testing / Justification | Test Data | Expected Result | | Actual Result |
|---|---|---|---|---|
| Testing the date picker to make sure any user input doesn't cause issue, as well as converting the date object into string formate (21/02/2020) for the database. | A few weeks in the future | Work fine, added to the database. Countdown to work. | | Did what I expect actually happen. Countdown displayed. |
| | 14/02/5646 | Could cause issues with countdown calculations. | | Countdown displays: "3625 years". Works fine, no errors. |
| | Todays Date (and waiting for a day to pass) | Could cause issues with countdown calculations. | | Current day - countdown displays: "Now 🎉"<br>Day after - Countdown displays: "Past 🕒".<br>All as expected. |
| | N/A, Impossible to add dates in the past. Therefore, "Rejected by the system". | Even if you could, countdown should just display its in the past. | | It does display its in past, no errors like expected.<br>Countdown displays: "Past 🕒". |

## *Testing Colour Picker*

| Testing / Justification | Test Data | Expected Result | | Actual Result |
|---|---|---|---|---|
| Testing the colour pickers which create the gradients. Trying with different colours to make sure gradients are made correctly. | Red and Blue | Gradient created successfully. | | *Gradient created successfully* |
| | Not changing colours | The preset white (primary) and grey are used. Set when variables are initialised. | | *Gradient created successfully* |

| Testing / Justification | Test Data | Expected Result | Actual Result |
|---|---|---|---|
| | Same exact colours (red and red). | Gradient created successfully. | ***Program crashes!*** Needs to be addressed. Possible SwiftUI bug. |

### *Fixing the same colour bug*

Instead of adding validation to the users side, as this would require an ugly error message to pop up if they selected the same colour. I can just change the colour when its being added into the database. The error only occurs when the colours are EXACTLY the same. Therefore I can change it minutely. The average person will not notice.

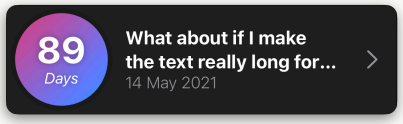
```
// 'addToDB'
// convert swiftUI color to string color array
let colorString1 = "\(data.colors[0].components.red),
                    \(data.colors[0].components.green),
                    \(data.colors[0].components.blue)"

// Changes to the 'addToDB' method
// adds 0.1 to other string to stop same color error
let colorString2 = "\(data.colors[1].components.red + 0.1),
                    \(data.colors[1].components.green + 0.1),
                    \(data.colors[1].components.blue + 0.1)"

// 'cardView'
// if colours are not the same, then use user selected colours
if (event.colors[1] != event.colors[0]) {
    Circle()
        .fill(LinearGradient(gradient: Gradient(
                                    colors: [event.colors[0], event.colors[1]]),
                                    startPoint: .topLeading,
                                    endPoint: .bottomTrailing))
} else {
    // bug fix selects same colour - BACKUP
    // use predefined colours
    Circle()
        .fill(LinearGradient(gradient: Gradient(
                                    colors: [Color.red, Color.orange]),
                                    startPoint: .topLeading,
                                    endPoint: .bottomTrailing))
}
```

As you can see above in the 'addToDB' section, I'm adding 0.1 to all RGB values, therefore always making sure when events are added to the database the colours are always different. Such a small value of 0.1 is impossible for the user to notice the difference.

As a backup, when the circle is created on the home screen, if it is found that the colours are the same before the gradient in the circle is made, then preset values are used instead, seen by the use of an if statement.

When the app was rebooted it fixed the bug, because the event with the same colours now just had the orange and red gradient like shown above. I can tell the slight change only when I set the colours both to black because the second colour in the gradient is slightly grey (again very hard to see - see right) . But if anything that is personally good as it gives that 3D effect.


### *Testing Text Input*

| Testing / Justification | Test Data | Expected Result | | Actual Result |
|---|---|---|---|---|
| This is the input where the user has most control, this is because they can type what they like in that text entry box. Making sure the database / app doesn't crash due to strange strings. Especially as I'm using SQL database certain strings can causes problems with the database | "Holiday 🌴☀️" | Inputed just fine. Displayed on the countdown card. | | The emojis didn't throw it, which is majorly important. Did exactly what was expected.<br>"Holiday 🌴☀️" Displayed. |
| | Nothing - "" | I added validation for this, the name of the event should come out with - "Untiled Event". | | Did exactly what was expected. The name of the event was "Untitled Event".<br><br>Even though an empty title should cause errors in the app this is more a UI design choice. |
| | "Bradley's Birthday" - the focus here is the apostrophe. | I'm not sure if this will cause an error or not, because the database under the surface is and SQL database. If not escaped normally this can cause major problems (SQL Injection). | | I personally didn't do any validation for this, but it seemed to work all fine. Must be all dealt with in the 'SQLite.swift' library.<br><br>The name of the event was "Bradley's Birthday". |
| | "What about if I make the text really long for no reason?" | Assuming that SwiftUI cuts it off automatically (...) if not then might have to be limited. | | As expected SwiftUI automatically limited the text.<br>  |
| | N/A - The only possible candidate could be limiting text size but I see no reason as shown above. | N/A | | N/A |

### *Fixing the update issue*

While testing this I noticed a major bug which was when you finish creating and event and the app takes you back to the main screen the event will not show on the main screen until the app is completely restarted. During testing, after adding an event I just rebooted the app to get round that. However it does need fixing.

Mentioned many times before if a state variable changes in a view the whole view will be updated. As well as this you can make functions run on changes of these variables.

Therefore to fix this issue I made a state variable on the main page called 'updateList' which just stores a Boolean which I can toggle.

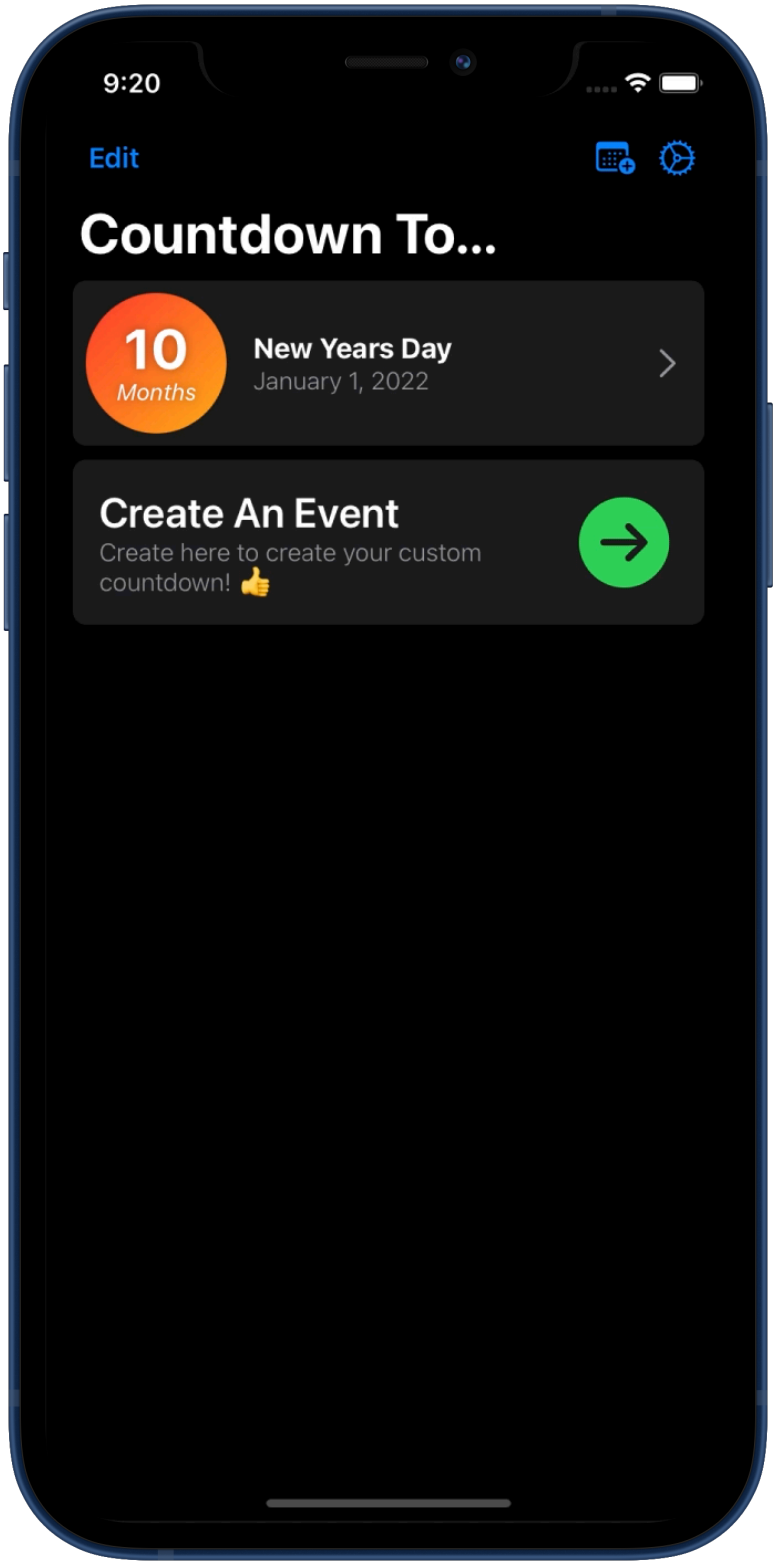
```
struct ContentView: View {

   @State var updateList: Bool = false

   var body: some View {
      ...
   }
   .onChange(of: updateList) { changed in
      print("updated required")
      events = databaseConnector().fetchEvents() ?? events
   }
}
```

From the code snippet above, the function 'onChange' is executed whenever updateList changes (in this case it is toggle). When it is executed, the method to fetch events is called and the returned value is stored in events. 'events' is a @State variable, therefore the view is reloaded with the new data.

```
// Navigation link updated to pass in 'updateList'
NavigationLink(destination: addEvent(updateList: $updateList))

// Added to 'addEvent' to allow update parent view, main screen.
@Binding var updateList: Bool

// When event added to database
if databaseConnector().addToDB(data: eventObject(name: eventTitle,
                               date: datePicked,
                               colors: [color1, color2])) {

   // If event added succesfully
   updateList.toggle() // Update parent view
   self.presentationMode.wrappedValue.dismiss() // return to parent view
   print("event added") // Print in console to allow debugging

}
```

This ‘updateList’ variable is needed to be toggled in the ‘addEvent’ view when the button to add an event is clicked. This can be done by using a @Binding, as explained above. I can just pass the @State in the main screen to the binding version in the ‘addEvent’ screen. Therefore it can be toggled when button is clicked.

***Click here to watch video***



***Relating back to the success criteria***

By relating back to the success criteria after every major segment allows me to make sure I keep on track and organise my development process.

***Event Creation - Success Criteria***

- Input name of event ✅
- Input date (time? ⛔) of event ✅
- Input two colours which will create a gradient background ✅
- Background image for event instead of gradient ⛔
- Repeating events (weekly, monthly, yearly) ⛔

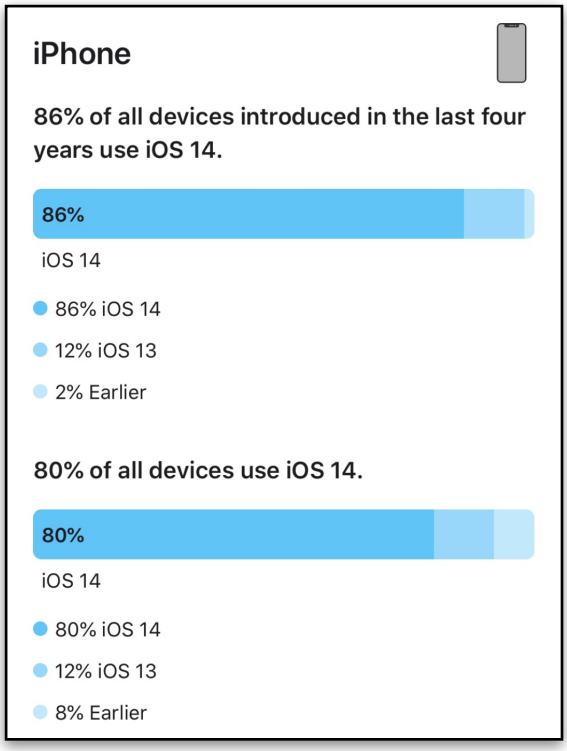
Obviously looking at that moment not all of it has been done, but what I have currently is a foundation so I can start to focus on the widget section of the app, which is its unique selling point.

These features stated above might get added further on in development depending on how things move forward, some not even on here may get added.

## Widget Integration

What you could argue is the most important part of this app, the widget, mainly because it's the unique selling point as well the main way people will view and interact with the app after events have been created.

Widgets are a new feature of iOS (14). Adoption rates, as mentioned in the analysis section, are very high for Apple products. Updates in some cases are done without the users knowledge overnight (if consent is given). This means there are no real reasons why you wouldn't update, apart from if you have jailbroken your iPhone (which is not very common). This is why as seen in the statistics *from Apple* (see right) **80%** of all iPhone run iOS 14.




As I am an official Apple developer, this means I get official betas when there are software updates. Personally I run betas of big version releases which are from months June till September (when the update is released to public). I started developing this early September ready for the release as this only works on iOS 14 due to the widgets.

### *WidgetKit*

This is the native library which is used to create widgets. Obviously this is going to be the greatest challenge as I have had no prior experience with this library. Furthermore as it was new documentation was sparse, with Apple and a few random YouTube videos being the only information and examples available.

## *Prerequisite*

Firstly, ⚠️WARNING⚠️, this section is very Apple. Related to how Apple do things and how this impacts development, and the user experience for the better.

### *Sandboxes*

One of Apple's main focus is privacy, this can be seen by the way apps are handled by the operating system. All apps are "sandboxed", this means that apps are restricted from accessing files stored by other applications or any other data stored on the phone. Each app has its own home directory to store files relevant to that app. You can see this above in the 'createTable' function or the code snippet bellow.

```
let path = NSSearchPathForDirectoriesInDomains(
    .documentDirectory, .userDomainMask, true).first!
```

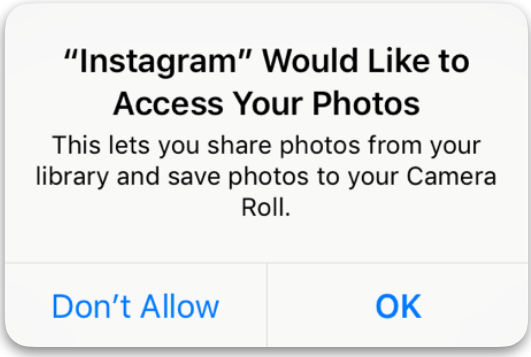
This allows me to get the path of the apps 'Documents' directory, which I can then use to create a database. This documents directory is obviously inside the "sandbox". Obviously I couldn't just change this directory and get inside another app. Even if I tried this is blocked by the operating system.

### *"What about when an app accesses my photos?"* 🤔

It would be correct to say that the photos apps data is outside a 3rd parties apps sandbox, so why can I access my photos in other apps such as Instagram, Snapchat, etc. This is due to entitlements and frameworks.

In this case the 3rd party app has the ability to access this data through a framework - 'PhotoKit'. A 'UIImagePickerController' class can be used that allows the user to pick the photo that they like from their library. When it is selected via the picker, that one picture that has been selected is returned via the 'didFinishPickingMediaWithInfo' function. Therefore the app never has direct access to the photo library as it is using a system built framework. The only data returned to the app itself

is the data selected. Furthermore, frameworks aren't allowed to be used unless you meet certain privacy requirements. One of these being supplying a description to the user of why you're allowing this app to use the framework (have access to your photos), this description is used when the 'requestAuthorization' function is called.




***Targets & Widgets***

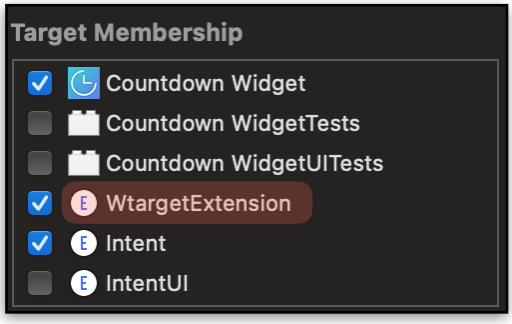
"A target specifies a product to build". Targets can be treated as a completely separate application; with their own entitlements, project files, sandbox, everything. Obviously a project can contain multiple targets. When a target is created it inherits Xcode's project build settings, however these can be overridden at the target level. "A target and the product it creates can be related to another target".

The 'widget extension' is a separate target from the main application so in the eyes of the operating system they are different applications.

Depending on how you view it and your workflow these targets make a lot of sense, especially as all applications (iOS, macOS, watchOS) can use the Swift programming language. For example: Microsoft could have one project for Microsoft Word and be able to make targets for iOS/iPadOS and macOS. These targets can share code files. This is via 'target membership'. This means that when code is compiled for a target, that file is included even if it's in another target.

***How does this relate to anything?***

The 'widget extension' is a separate target from the main iOS app. I needed to allow the widget target access to the 'databaseConnector' file in the iOS target, access given via 'target membership'.



## *Prerequisite*

At this point I had no knowledge of how targets were treated as separate applications with different sandboxes, therefore different 'Documents and Data'.

### *How I learned about everything above?*

Now I have everything in place I can start to create a simple widget (I will go into more detail about this later). Firstly I made a simple widget view for test, as again 'WidgetKit' is completely new to me at this point. After doing the stupid "Hello, World" stuff I decided I should flesh out the widget.

Just to prove fetching data from the database worked correctly I added an 'onAppear' to the view, which fetched the data using the functions in 'databaseConnector'. When I executed the 'WtargetExtension', a widget appeared on the simulators home screen. In the console, which as I am running the 'WtargetExtension' itself. The console is for the widget only. It showed this error:

```
<Console - WtargetExtension (Widget)>
2021-02-28 17:27:30.713132+0000 WtargetExtension[2632:87680] [widget] No intent in
timeline(for:with:completion:)

# This is becuase 'events.sqlite3' table does not exist, widget has own "sandbox" #
no such table: events (code: 1)
```

Why was there no table? As always in programming, the computer is only following what you say, you're the problem. There wasn't an 'events' table. Completely different directories for the main app (where the 'events' table is stored), and the widget.

### *App Groups*

An app group allow me as well as other things to create a shared container. This shared container can be accessed by all targets that have that container identifier in its entitlements. I called the app group: "group.widgetShareData.countdown", because its purpose is to allow me to share data with the widget. The 'addToDB' method had to be updated to use the app groups shared container path for the database

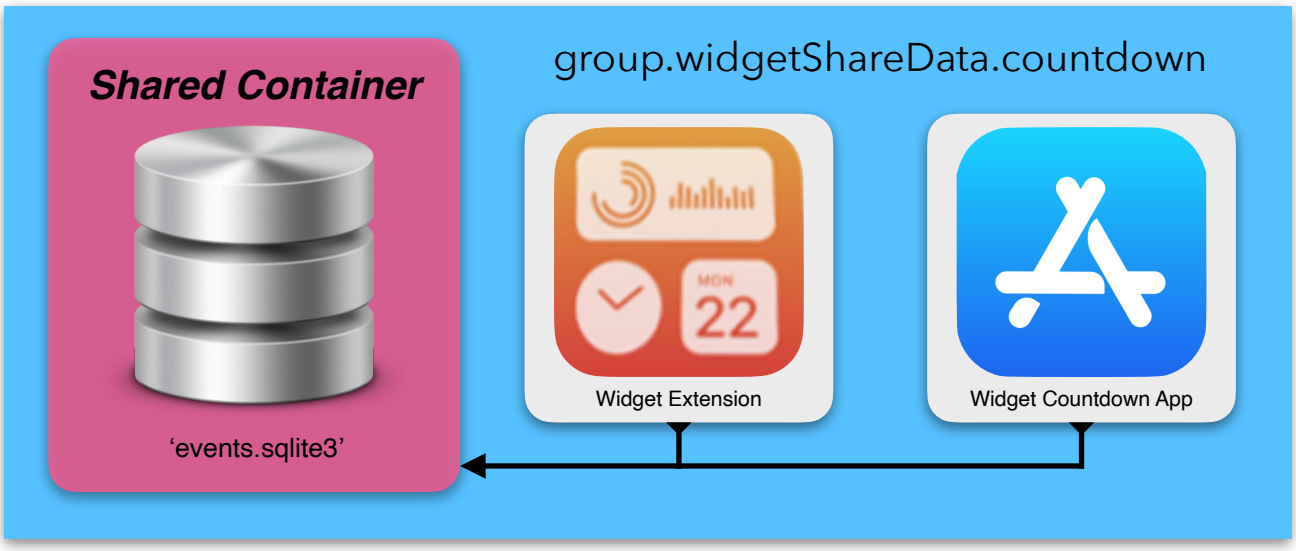
file and not the standard 'Documents and Data' shown when I first programmed it.

```
public class databaseConnector {

    // the path to the shared container directory
    let path = FileManager.default.containerURL(forSecurityApplicationGroupIdentifier:
                                        "group.widgetShareData.countdown")!.absoluteString

    func createTable() -> Bool {
        do {
            let db = try Connection("\(path)/events.sqlite3")

            // create table
            try db.run(eventsTable.create { t in
                t.column(id, primaryKey: true)
                t.column(eventName)
                t.column(date)
                t.column(color1)
                t.column(color2)
                t.column(bgIMG)
            })

            //table make successfully
            return true
        } catch {
            //idk what went wrong
            return false
        }
    }
...
```

After running the same code again, the console printed out the event objects in the database successfully which meant the database was now stored in the shared container and both targets (widget and main app) could access said data.

```
<Console - WtargetExtension (Widget)>

[Countdown_Widget.eventObject(id: D93E5711-DB43-45D2-B3C6-FD5D72B037BA, name: "New Years
Day", date: 2022-01-01 00:00:00 +0000, colors: [#FF3B30FF, #118AF19FF], bgIMG:
Optional(""), databaseID: Optional(1))]
```

Below is diagram of what's happening:



### *IntentConfiguration*

The widgets need to be configured for the app using the 'WidgetConfiguration' protocol. This can contain multiple widget, this gives the ability for apps to have many widget types that do different things. There are two types of widget: "StaticConfiguration" and "IntentConfiguration".

As the names suggest. A static widget means it has "no user-configurable options". The other uses SiriKits intents, to allow widgets to have "user-configurable options". This is obviously what I require as the user will have to select which countdown they want to be displayed.

```
@main
struct CountdownWidget: Widget {
    private let kind: String = "Countdown Widget"

    // Setting up widgets in your app (app can have multiple widget types)
    public var body: some WidgetConfiguration {
        // A widget that uses an intent to provide user-configurable option
        IntentConfiguration(kind: kind, intent: ConfigurationIntent.self,
                            provider: Provider()) { entry in
            // A view that has an entry passed into it created by the 'provider'
            WtargetEntryView(entry: entry)
        }
        .configurationDisplayName("Countdown Widget") // Name of widget
        .description("Select your chosen countdown in widget settings...")
        .supportedFamilies([.systemSmall, .systemLarge])
        // Widget sizes allowed for this widget ^
    }
}
```

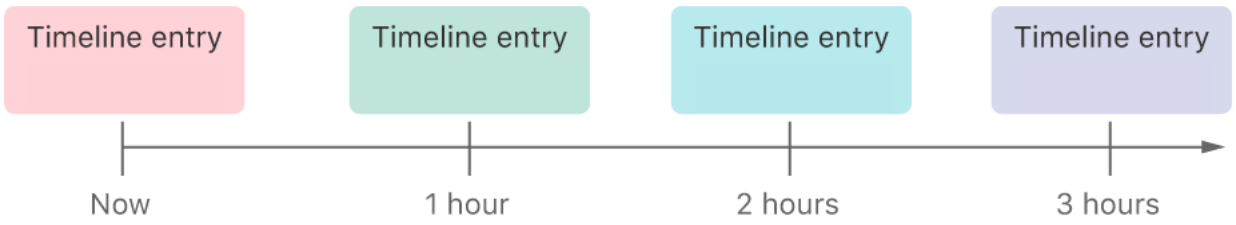
The code snippet above is what configures the widget. Inside the 'body' I could put as many configurations (static / intent) as I wanted to allow the app to have many widgets.

The 'IntentConfiguration' has a 'provider'. This 'provider' contains inbuilt functions to create and provide the widget ('WtargetEntryView' in this case) with data in the form of a timeline entry, as per the name provider.

After that there are just modifiers which specifies things about that widget, for example the sizes the widgets can be displayed.

### *Timeline*

Widgets use a timeline, this allows a widget to run as smooth as possible, widget states are generated in advance. The best way to explain is a calendar widget. The calendar will know when events in the day take times wise. Therefore timeline events can be created for those times. A timeline can be created for the entire day (obviously this can be updated if a calendar event is updated, deleted, removed, etc).




The calendar widget now already knows when to update. For certain applications this can be used and is very efficient.

### *TimelineEntry*

A timeline is created from an array of timeline entries, this protocol specifies the timestamp that data is useful for the widget. When you define a struct that conforms to 'TimelineEntry', you can also define other data that the widget will use. In this case the event datatype I created earlier.

```
// Timeline entry that contains extra data
struct EventTimelineEntry: TimelineEntry {
   public let date: Date // this is to conform to the protocol
   public let event: eventObject
}
```

### *Designing the widget*

The widget is designed the exact same way as any other view. Therefore nothing much to explain, to create the widget though I did create elements of the widget as separate views.

Just as shown in the configuration, an entry is passed in and because that timeline entry also contains the custom data 'eventObject' (how that data gets there will be explained later), it can be used inside of this

```
struct WtargetEntryView: View {
    var entry: Provider.Entry

    var body: some View {
        ZStack {
            if (entry.event.colors[1] != entry.event.colors[0]) {
                RoundedRectangle(cornerRadius: 0)
                    .fill(LinearGradient(gradient: Gradient(colors: [entry.event.colors[0],
                                                                     entry.event.colors[1]]), startPoint: .topLeading,
                                                                     endPoint: .bottomTrailing))
            } else {
                //bug fix if user selects same colour
                RoundedRectangle(cornerRadius: 0)
                .fill(LinearGradient(gradient: Gradient(colors: [.red,.blue]),
                                                                  startPoint: .topLeading, endPoint: .bottomTrailing))
            }
            HStack {
                VStack(alignment: .leading) {
                    EventTitleView(entry: entry)

                    EventCounterView(entry: entry)

                    Spacer()

                    EventDateView(entry: entry)
                }
            }
            .padding(10)

            Spacer()
        }
    }
}
```

view. The background of this
widget, just like the circle in the app, is a gradient which colours are customisable by the user. So via the entry I can access the event data and therefore the colours for the gradient - “entry.event.colours[1]”. The gradient background also uses the same validation (if statement) as the circle making sure the colours are not the same so the widget does not crash, if they are the same some default values, red and blue, are used.

On top of the gradient is a stack of views (VStack) which contains the elements of the countdown: title of event, the countdown itself and and the date that countdown is set for.

### *Widget Elements*

I have separated these into separate views as it is a recommended practice as well as maintainability for the future. If for example I wanted to add a new type of widget I can reuse some / all of these elements. As well as making it easier to make changes in the future.

All of these elements pass in the timeline entry so they can use the relevant data, which is the eventObject at this point, this could be expanded on in the future. Which again reiterates the point of modularity to allow changes in the future.

## EventTitleView

This is the most simple view, which is just a text view with modifiers to display the event name.

```
struct EventTitleView: View {
    var entry: Provider.Entry

    var body: some View {
        Text(entry.event.name)
            .font(.title2)
            .bold()
            .padding(2)
            .foregroundColor(entry.textColor)
    }
}
```

## EventCounterView

This view actually displays the countdown, for example: “26 days”. This view can just reuse a function. Therefore I can just pass in the date object - “entry.event.date” - and it will return the relevant string. Furthermore ‘??’ are used in the text view as if there is any problem with the data that’s returned a default string (“Error”) is used instead to stop the app from crashing, this is also know as “unwrapping an optional”.

```
struct EventCounterView: View {
    var entry: Provider.Entry

    var body: some View {
        // same function as created earlier
        let dateStringData =  DateConverter().DateToString(dateObject: entry.event.date)
        // the '??' is a default value shown if the optional value can't unwrap
        Text("\(dateStringData["value"] ?? "Error") \(dateStringData["type"] ?? "")")
            .font(.title3)
            .fontWeight(.light)
            .foregroundColor(.white)
            // customising the text view with modifiers
    }
}
```

## EventDateView

This view displays the actual date of when the countdown reaches it end, for example: “28th Mar 2021”. It is being returned in that way by the built in ‘DateFormatter’ because it is using the ‘.medium’ date style.

Just like in the one before the date object of the countdown stored in the timeline entry, “entry.event.date”, is used.

```
struct EventDateView: View {
    var entry: Provider.Entry

    var body: some View {
        Text(dateToCalString(dateConvert: entry.event.date))
            .font(.subheadline)
            .fontWeight(.ultraLight)
            .foregroundColor(.white)
    }

    // function that takes in date object and returns date string that follows the
    // .medium date style (eg: 28 Mar 2021)
    func dateToCalString(dateConvert: Date) -> String {
        let dateFormate = DateFormatter()
        dateFormate.dateStyle = .medium
        let stringDate = dateFormate.string(from: dateConvert)

        return stringDate
    }
}
```

That explains fully how the widget is designed and how it uses the data from the ‘Provider’.

### *The Provider*

The provider, which is this case conforms to the protocol ‘IntentTimelineProvider’ has all the inbuilt functions to create a timeline for widgets. But firstly before all the complicated stuff. There are 2 functions: “getSnapshot” and “placeholder” which these functions are supposed to return a timeline entry with default data which can be used when the widget is being added by the user / data for the widget is being loaded. You can see on the picture on the right, the provider returning sample data for the widget selection process in this case a holiday happening in the future.


Below is the code snippets of the “placeholder” and “getSnapshot”:

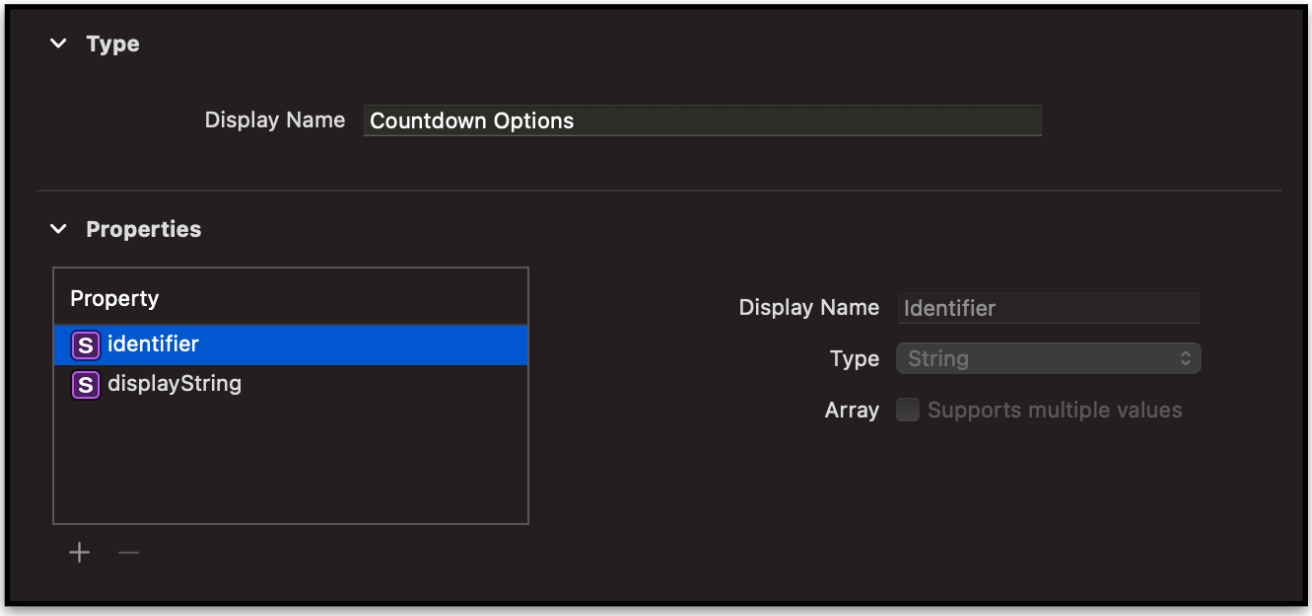
```
struct Provider: IntentTimelineProvider {

    func placeholder(in context: Context) -> EventTimelineEntry {

        return EventTimelineEntry(date: Date(), event: eventObject(name: "Holiday", date:
                            Date(timeIntervalSinceNow: 7884000), colors: [.red,.orange]))
    }

    func getSnapshot(for configuration: ConfigurationIntent, in context: Context,
                                completion: @escaping (EventTimelineEntry) -> Void) {

        let entry = EventTimelineEntry(date: Date(), event: eventObject(name: "Holiday",
                    date: Date(timeIntervalSinceNow: 7884000), colors: [.red,.orange]))

        completion(entry)
    }
...
```

The functions are returning a 'EventTimelineEntry', defined above. Which also contains and 'eventObject' with sample data. This sample data containing: a title, the gradient colours, and the date the event will take place which is set to 3 months from "now" (when this is returned).

### *Intent*

Before going any further with the provider, I need to set up the intent which allows the widget to be user configurable. On the right is an example of a user configurable setting, in this case the weather widget. For my widget I need the user to be able to select from the countdowns they have made.


In the intent configuration in the widget I can create a "custom type":



To show the event in the configuration I need to override the default implementation of the intent.

```
class IntentHandler: INExtension {

    override func handler(for intent: INIntent) -> Any {
        // This is the default implementation.  If you want different objects to handle
        // different intents, you can override this and return the handler you want for
        // that particular intent.

        return self
    }
}

extension IntentHandler: ConfigurationIntentHandling {
    func provideCountdownChoiceOptionsCollection(for intent: ConfigurationIntent, with
        completion: @escaping (INObjectCollection<CountdownOptions>?, Error?) -> Void) {

        // init an array of countdown options, defined in intent definition
        var eventOptions = [CountdownOptions]()

        let events = databaseConnector().fetchEvents()! //fetch all events from db

        // for all events in the database create an CountdownOptions object with
        // the database id of that event, as well as the events name as the display name
        events.forEach { event in
            let IntentObject = CountdownOptions(identifier: "\(event.databaseID!)",
                                                display: event.name)

            // appened this object to the eventOptions array
            eventOptions.append(IntentObject)
        }

        completion(INObjectCollection(items: eventOptions), nil)
    }
}
```

In the above code, an array of ‘CountdownOptions', the custom type I created above, is initialised. Then all the events from the database are fetched and are gone though using a foreach. Using the event data fetched ‘CountdownOptions’ is made using the database ID as the identifier and the event name as the display name. I can use the configuration intent later in the provider to customise the widget.

***The Provider cont.***

After all this build up I can now generate a timeline for my widget!

```
func getTimeline(for configuration: ConfigurationIntent, in context: Context,
                 completion: @escaping (Timeline<EventTimelineEntry>) -> Void) {

    print("refresh")

    // when the widget should next refresh. One min in the future
    let refreshDate = Calendar.current.date(byAdding: .minute, value: 1, to: Date())!

    // get the indentifier from the intent configuration
    // the indentifier in this case in the id of that events record in the db
    let eventID = configuration.CountdownChoice?.identifier ?? "0"

    // custom fetch function, to fetch an event WHERE id is the event id
    let eventData = databaseConnector().fetchEventsID(filterID: Int64(eventID) ?? 0)

    let entry = EventTimelineEntry(date: Date(), event: eventData)

    let timeline = Timeline(entries: [entry], policy: .after(refreshDate))

    completion(timeline)
}
```

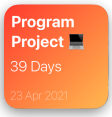
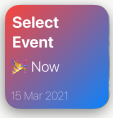
This function is executed every time a timeline needs to be generated. When a timeline is generated is defined by the “refresh policy”. In this case a timeline is generated after the refresh date is reached. Which is one minute after this timeline is generated. In simple terms a timeline should be generated every minute.

*When a timeline is generated:*

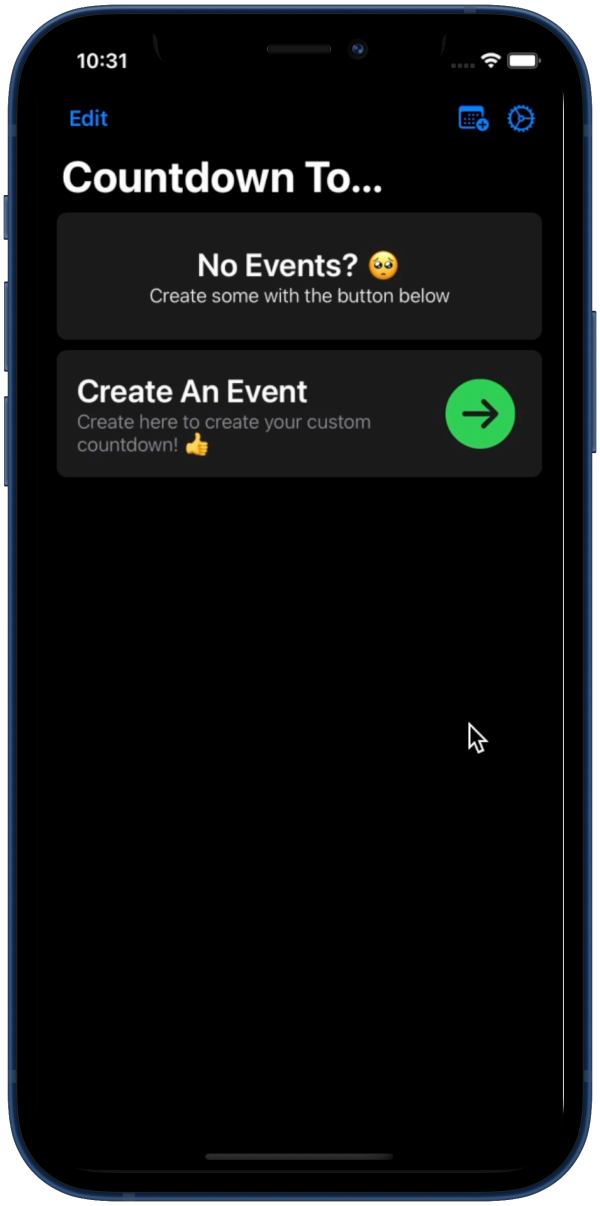
- Configuration intent is passed into the function, from the configuration intent the ‘CountdownChoices’ identifier (which is the events record ID in the db) is stored in a variable.
- With a customised fetchEvents function that fetches only the event WHERE id is equal to the ‘CountdownChoices’ identifier.
- An entry can then be made with the current date and eventData fetched.
- Now a timeline can be created with only that entry in it, and its refresh policy as mentioned above.

***The widget is finally made!***

# Widget testing

| Testing / Justification | Test Data | Expected Result | | Actual Result |
|---|---|---|---|---|
| The widget needs testing to make sure, that everything updates correctly with the timeline, as well as handling the custom user configuration (intent) | Trying different countdowns | The countdown selected loads in | | Countdown loads in and displayed correct   |
| | Setting no default countdown | Countdown name: “select event” | | Countdown loads in and displayed correct   |
| | Clicks onto widget | App opens to main screen. | | App opens successful |
| | **N/A** | | | **N/A** |

***Click here to watch video***



***Relating back to the success criteria***

Again just like before by relating back to the success criteria it allows me to make sure I keep on track and organise my development process, making it easy to see what I works needed.

***Widget System - Success Criteria***

- Diffrent sized widget (.systemSmall, .systemMedium, etc) ✅
- Customise the widget ✅
  - Different colour text ⛔
  - Organise / Remove widget elements (count, date, name) ⛔
- Image as choice as widget background ⛔
- Updates daily, as countdown decreases ✅

Just like the last time, not all of them have been done at this moment but this can change in the future.

But looking at the current apps state is everything which is required for a basic countdown app is there. Everything from now on is to add the USPs for users.

## Miscellaneous

This section is me starting to add the finishing touches to the project.

### *Photo Background*

This will allow the user to customise their widget with a background that they can choose from their photo library. SwiftUI is a new language only being out for just over a year at the time of writing/development. Therefore the extensive feature set of iOS hasn't been fully implemented into the framework unlike UIKit.

That means even some key features are not natively available. One of those that I use been "UIImagePickerController" (PhotoKit). However SwiftUI has a solution to this problem: UIViewControllerRepresentable.

This allows you to use UIKit directly inside SwiftUI therefore giving you the best of both worlds. The mature, highly documented UIKit and the new, highly optimised, declarative based system of SwiftUI.

```
struct Picture: View {

    @State private var showImagePicker: Bool = false

    @Binding var pickedPhotoData: String


    var body: some View {

        Button(action: {
            //clicked change image button
            if pickedPhotoData == "" {
                showImagePicker.toggle()
            } else {
                pickedPhotoData = ""
            }
        }) {
            ...
            // Designing button just like before
            ...
        }
        //show image picker, ether camera roll or camera depending on action sheet
        .sheet(isPresented: $showImagePicker, content: {
            ImagePicker(pickedPhotoData: $pickedPhotoData,
                        isShowingPicker: $showImagePicker)
        })
    }
}
```

The code snippet on the previous page, shows the code for the button on the 'addEvent' page, this button is designed exactly the same as all the others. However this one brings up a sheet when clicked, showing the view 'ImagePicker', which passes in the binding 'pickedPhotoData'. This binding relates to a state variable in the parent view 'addEvent'. So as mentioned many times before. Whenever the binding is changed that is propagated up to the state variable and visa versa.

```
//The image picker it self
struct ImagePicker: UIViewControllerRepresentable {

    // the vars passed in
    @Binding var pickedPhotoData: String

    @Binding var isShowingPicker: Bool

    // where the user is getting image data from, in this case their photo library
    var sourceSelected: UIImagePickerController.SourceType = .photoLibrary

    // Creates the custom instance
    func makeCoordinator() -> ImagePickerCordinator {
        return ImagePickerCordinator(isShowingPicker: $isShowingPicker,
                                     pickedPhotoData: _pickedPhotoData)
    }

    //Image picker is only UIKit not SwiftUI therefore need to use
    //UIViewControllerRepresentable
    func makeUIViewController(context:
        UIViewControllerRepresentableContext<ImagePicker>) -> UIImagePickerController {

        let picker = UIImagePickerController()
        // set the delegate to the custom instance
        picker.delegate = context.coordinator

        // when user has selected their image, allow basic editing (cropping)
        picker.allowsEditing = true

        // set source of the picker controller
        picker.sourceType = sourceSelected

        // return UIImagePickerController with set attributes
        return picker
    }

    func updateUIViewController(_ uiViewController: UIImagePickerController,
        context: Context) {
        //not needed but protocol so required anyway
    }
}
```

Above is the actual code related to the image picture, this set up all the relevant settings to create the UIKit view. Furthermore a coordinator has to be used. This is a custom instance that you use to communicate changes from your view controller to other parts of your SwiftUI interface. In this case the coordinator becomes the delegate. A delegate (as described simply by HackingWithSwift) "a delegate is any object that should be notified when something interesting has

happens". This makes it clear the difference between SwiftUI and UIKit, delegates don't really exist in SwiftUI.

```
class ImagePickerCordinator: NSObject, UINavigationControllerDelegate,
        UIImagePickerControllerDelegate{

   // vars passed in AGAIN (see a trend yet :))
   @Binding var pickedPhotoData: String

   @Binding var isShowingPicker: Bool

   init(isShowingPicker : Binding<Bool>, pickedPhotoData: Binding<String>) {
      _isShowingPicker = isShowingPicker
      _pickedPhotoData = pickedPhotoData
   }

   // function that runs when image selected, as per name 'didFinishPickingMedia'
   func imagePickerController(_ picker: UIImagePickerController,
         didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey : Any]) {

      // store the edited image in a variable
      let imageSelected = info[UIImagePickerController.InfoKey.editedImage] as! UIImage

      // gets the UIImage and converts it into a Base64 string
      let imageData = imageSelected.pngData()
      let strBase64 = imageData!.base64EncodedString(options: .lineLength64Characters)

      // set the binding pickedPhotoData to that new Base64 string
      self.pickedPhotoData = strBase64

      //close the image picker
      self.isShowingPicker = false
   }

   func imagePickerControllerDidCancel(_ picker: UIImagePickerController) {
      isShowingPicker = false
   }
}
```

This above is the coordinator (delegate) of the created UIImagePickerController, this is a method called 'didFinishPickingMedia' which is a method of UIImagePickerController. This function, as the name suggests, is called when the user has selected an image. One of the parameters of this function being the said image.

### *Storing photos in DB*

There are two ways I could store this type of data in the SQLite database: reference or actual value.

### Actual Value

After reading this **Stackoverflow question** on storing images in SQLite, I decided to go with the suggested answer, which is to convert the image to Base64 string and to then store that string. Even though this leads to the string being quite large, knowing the processing

power and efficiency of iOS this shouldn't be a problem. This is why you see above in the code snippet. After storing the image selected in a variable - 'imageSelected', I then convert it into a Base64 string and set the value of 'pickedPhotoData' to said string.

Just for reference that photo stored in the database ended up being: ***2016898 characters long***.

***Photo metadata:*** 1,466,835 bytes, 1171 × 1170 resolution.

## Referencing

This is the most used method of "storing" (using the word "store" loosely) and image in an SQL database. Which is to store the image at a path in the applications/websites directory. Then in the database store the path of the image.

This would save space in the database and probably increase fetch speed (probably not noticeable anyway). However I didn't like the fact they are stored separately, so if in the future I added an "export events" feature, or sharing events, it would be harder to achieve. As storing the images data in the database means all the events data are stored in that single record. Base64 is also universal, blob isn't.

As I already planned this, above in the 'addToDB' code I had already created a "bgIMG" field. Therefore nothing needed to be changed apart from passing in the image data to the function - 'pickedPhotoData'.

### *Added the photo to the widget*

This was easy to add, all that was needed to be done check if image data is available, and if so add it above the rest of the widget in the ZStack. This is seen in the code below.

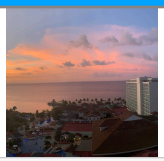
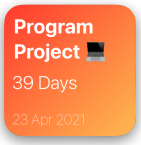
```swift
if (entry.event.bgIMG != "") { // if photo data in event data
    // convert the Base64 string into 'Data' datatype.
    let dataDecoded = Data(base64Encoded: entry.event.bgIMG!,
                           options: .ignoreUnknownCharacters)

    // assume if not nil decode was succesful
    if (dataDecoded != nil) {
        // the swiftUI image type can't currently take 'Data' as parameter
        // Therefore I can convert a UIImage that does allow you to use data
        // Convert said UIImage into SwiftUI image.
        let decodedimage = Image(uiImage: UIImage(data: dataDecoded!)!)

        ZStack {
            decodedimage //SwiftUI Image
                .resizable() // allow it to resize if needed
        }
    }
}
```

It first checks if image data is available, is the string empty. If it's not then that string can be decoded into the SwiftUI image type, see comments for more detail. This code was also added to the main screen to card view.



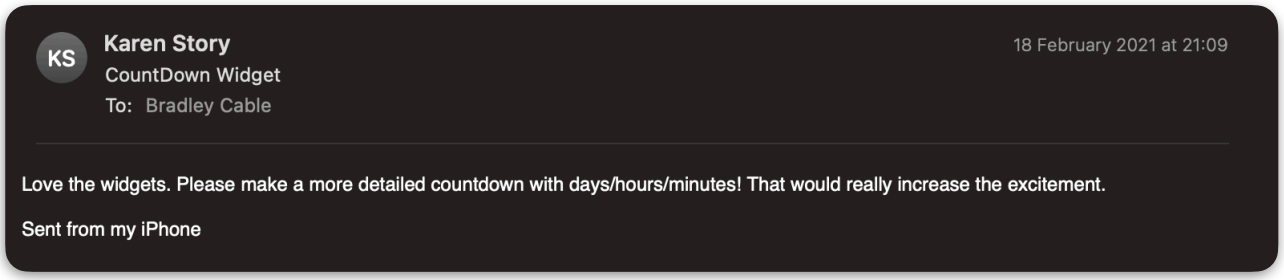
***Testing widget feature***

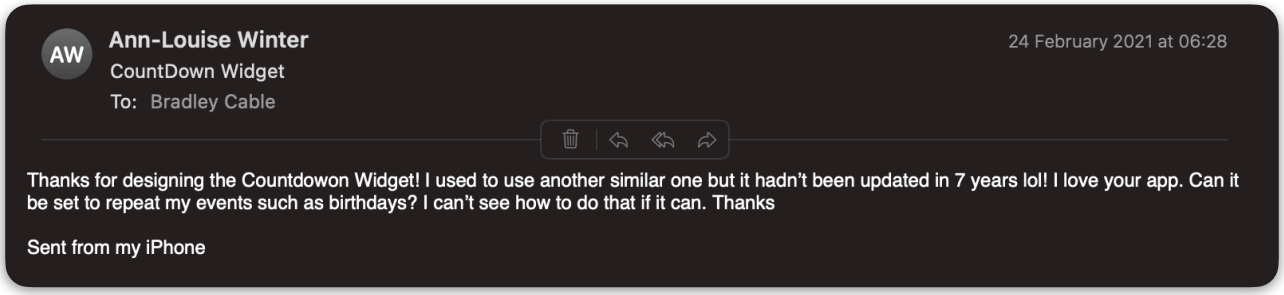
Now this doesn't require a lot of testing because a lot of validation is handled by Apple frameworks, for example if the image needs downloading from iCloud and it fails that is already handled. The only thing I could see failing is the Base64 conversions. However how it converts I can't change. But as said above I do some simple validation at last making sure that the 'Data' object isn't "nill" before doing anything else.

| Testing / Justification | Test Data | Expected Result | | Actual Result |
|---|---|---|---|---|
| To make sure that when adding a background image instead of a gradient that different image type / lack of image cause any problems, inside the apps card view or widget. |  | The countdown works as expected with the selected image as background. | | Countdown loads in and displayed correct.   |
| | Completely transparent image. (See here) | The widget/card view, is completely black as there is nothing to show only “Alpha”. | | The background of the widget/ card view turned out to be completely black, I don’t know if that is usual of transparent images. I asked Jamie what he would of expected, he also said black. Anyway, the app and widget had no problems no crashing |
| | No image | The countdown works as expected with the gradient as the background. | | Countdown loads in and displayed correct.   |
| | **N/A** | | | **N/A** |

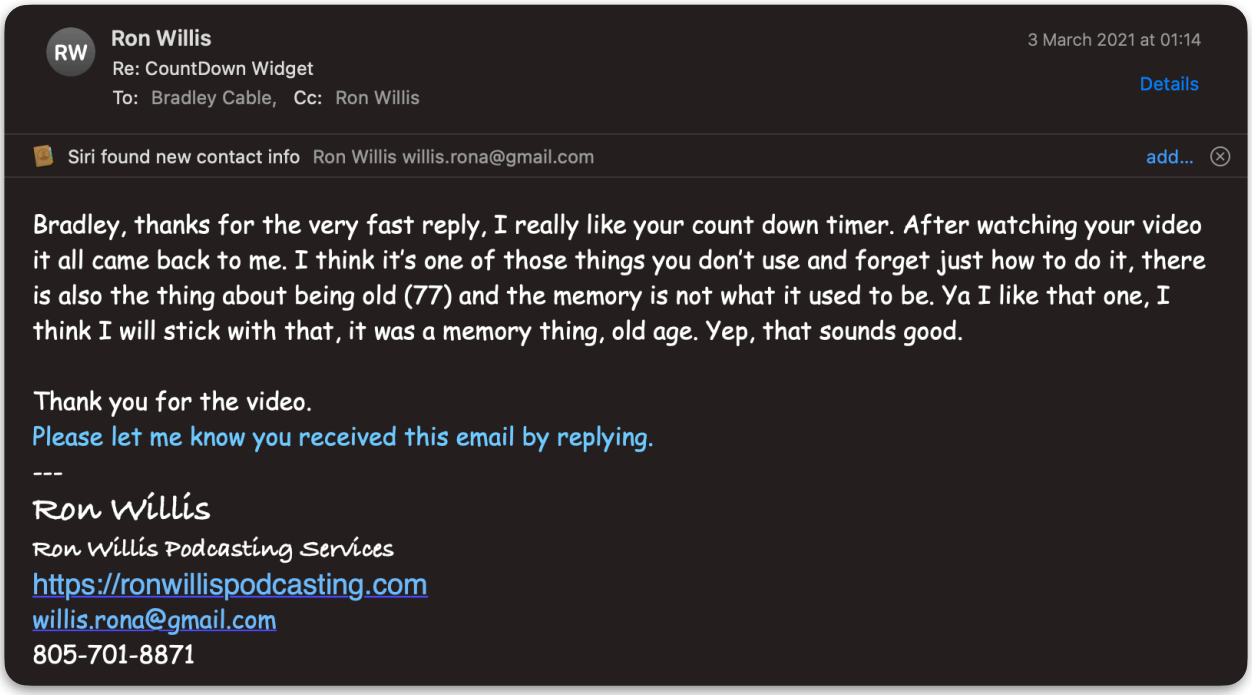
## User Feedback / Usability Testing

After realising it onto the App Store I started to receive emails from users, I didn’t expect lots of people download in the early days (more detail on this later), so the general public became testers by accident!



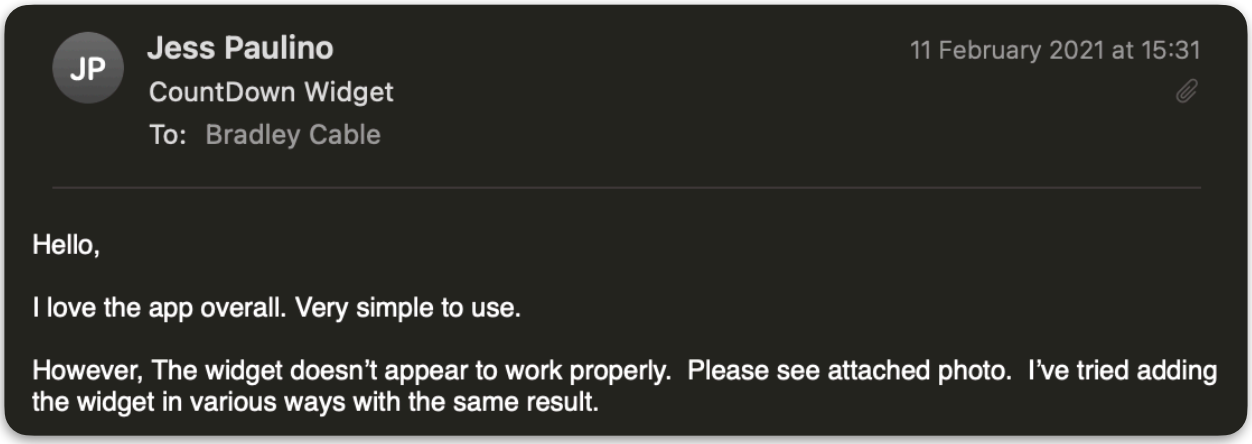



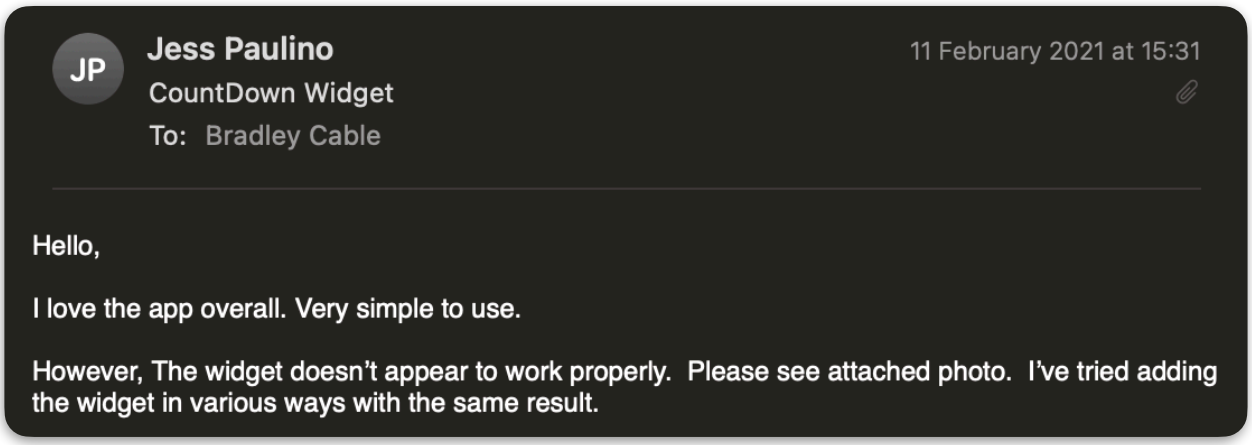

Small feedback like this shows the demand for certain features, even some that I have not considered, for example, allowing the user to select how the event is counted down: days, months, and years.




This email from Ron, 77, highlights to me the range of the demographics that use my app. In my analysis I mentioned about the demographics of  iPhone's (15-18). Which really shows how I didn't take into account the high age ranges, and how they may use the app. Even though me personally, the developer, might know how to add a widget, doesn't mean that everybody would. Form just looking I still have in my inbox I have five which relate to confusion on how to add widgets to how screen, this shows a lack of consideration on my part.
Now due to Apple privacy, I can't use the analytics of the app to tell the age demographics of users as goes against Apple's policy unless I collected the data myself and declared that I was using it. However even that is not allowed as data has to be relevant (5.1.1 iii), and as the age isn't relevant for the apps function, asking for it in a future update would mean my app would be rejected.

> “5.1.1, iii) **Data Minimisation:** Apps should only request access to data relevant to the core functionality of the app and should only collect and use data that is required to accomplish the relevant task. Where possible, use the out-of-process picker or a share sheet rather than requesting full access to protected resources like Photos or Contacts.” - Extract from App Store Review Guidelines

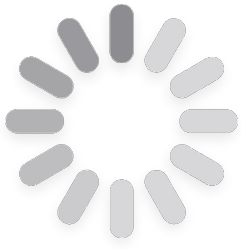



As well as features and recommendations. I also received many bug reports. Focusing on the last one for the document. This was a bug report that said the user had problems selecting the event they wanted to display on the widget in the widget setting.

At first it was hard for me to reproduce the bug so I asked for more info. They responded saying, they had around 20 events in the app. So I assumed that is was an issue relating to the amount of widgets.

I also tried adding some titles with weird text and symbols. For example a mix or Arabic (right to left text) and English (left to right text). However there was still no issues (probably due to the amazing text handling of iOS and unicode).



I finally worked out what the issue was when I added background images to all the events. All of a sudden there was a loading symbol when I clicked on 'Countdown Choice'. So obviously the 'Intent' was not

happy to deal with the amount of data that was being fetched from the database, with so many images being stored in Base64 strings.

Some basic maths suggests if each image is about two million characters long (2,000,000).

***25 events * 2,000,000 = 50,000,000 characters in data!***

No wonder there was a problem… 😅

Now you could argue, that was a bit of inefficiency on my part, but the main app still loaded fine. Furthermore the size of the database file that stored all the events, is in the single digit megabytes, which is what you would expect if you were storing a few images and some text normally. So my assumption what caused the issue, is some kind of limitation hard coded into a widget to make sure that home screen widgets don't take up too much processing time and cause any form of lag.

## Taking action

I can now take action on some of the feedback I received in the email add or fixing parts of my app…

### Widget settings loading

So just repeating what the problem is, because when you select an event in widget settings it uses the 'fetchEvents()' function to fetch all the events in the data. This is all the data. So in SQL it would be using the wildcard "*". Even though it wouldn't need all this data to display a list of events in widget settings. All that's needed is the ID of the record in the database and the event name itself to put in the list.

So just as simple as the problem is there was a simple solution. I added a new method the the 'databaseConnector' class called "fetchEventWidgetChoice()". This again selects all records but only the fields necessary, in this case ID and eventName. The code below shows this new method…

```
func fetchEventWidgetChoice() -> [eventObject]? {
    do {
        let db = try Connection("\(path)/events.sqlite3")
        // init an arry that contains event object
        var tempArray = [eventObject]()

        // select only the two fields
        for event in try db.prepare(eventsTable.select(id, eventName)) {
            // loop though all the events
            // add to temp array
            tempArray.append(eventObject(
                                name: "\(event[eventName])",
                                databaseID: event[id]))
        }

        // return the finshed array
        return tempArray
    } catch {
        // used for debuging
        print(error)
        return nil
        // return nil to show error
    }
}
```

Now I just had to change the function used in the “IntentHandler” and then it all worked smoothly.

## Adding more customisability

As per the first email, that suggested being about to show the countdown in different formats. Now obviously that is already possible to some extent as it is “dynamic". It will first countdown with years, months, then days. Showing the one that’s most relevant depending on how many days are next.

However some users might only want it displayed in months, or day, etc. To reflect this I’ve had to make changes to the function that works out what to show. Whilst I was adding this feature, I also decided to crest a feature that feature to change text colour of the widget, which will help if the user uses and image / gradient which white text doesn’t work on and allows it to be more personal.

To add more to the widget setting I had to add enums to the Intent. Enums defines a common type for a group of related values and enables you to work with those values in a type-safe way (preventing silly errors).

I added a colour type and counter type enum to the intent. The colour type contains 10 colours and unknown (which will just default to white in the code). 10 colours because that how many preset Swift colours there are.

The counter type enum contains three options: days, months, years, dynamics and unknown (which will default to dynamic).

```
func DateToString(dateObject: Date, type: CounterTypeEnum) -> [String: String] {
// passes in type that conforms to 'CounterTypeEnum' (.days, .months, .years, .dyanmic)
...
   // if .dynamic or .unknown value then use what was being used before
   if (type == .dynamic || type == .unknown) {
      // see above for a more detailed explanation
      if toDays.day! > 99 {
         if toMonth.month! > 12 {
            returnResult["value"] = String(toYear.year!)
            returnResult["type"] = "Years"
         } else {
            returnResult["value"] = String(toMonth.month!)
            returnResult["type"] = "Months"
         }
      } else {
         returnResult["value"] = String(toDays.day!)
         returnResult["type"] = "Days"
      }
   // if the dynamic display method isn't used…
   } else {
      // use the relvent conversion and string for each counter type
      if type == .days {
         returnResult["value"] = String(toDays.day!)
         returnResult["type"] = "Days"
      }
      if type == .months {
         returnResult["value"] = String(toMonth.month!)
         returnResult["type"] = "Months"
      }
      if type == .years {
         returnResult["value"] = String(toYear.year!)
         returnResult["type"] = "Years"
      }
   }
...
}
```

The code snippet above shows the changes to the 'DateToString' method. Now using the 'CounterTypeEnum' in the parameter of the function. When the actual type is passed in as an argument that is used to work out what is returned. On the main page in the app the ".dynamic" type will always be used.

Now the actual widget had to be changed to take this into account. These variables needed to be added to the "EventTimelineEntry" struct.

These changes are shown below:

```
// Timeline entry that contains extra data
struct EventTimelineEntry: TimelineEntry {
   public let date: Date // this is to conform to the protocol
   public let event: eventObject
   public let textColor: Color
   public let counterType: CounterTypeEnum
}
```

Added a function to the widget file that takes in the intent configuration as in input, then uses the option selected for text colour in a switch to return the relevant SwiftUI Color.

```
func textColorConvert(for configuration: ConfigurationIntent) -> Color {
   // Converts the text colour enum choice to SwiftUI Color
   switch configuration.TextColor {
      case .unknown: return Color.white
      case .white: return Color.white
      case .black: return Color.black
      case .blue: return Color.blue
      case .gray: return Color.gray
      case .green: return Color.green
      case .orange: return Color.orange
      case .pink: return Color.pink
      case .purple: return Color.purple
      case .red: return Color.red
      case .yellow: return Color.yellow
   }
}
```

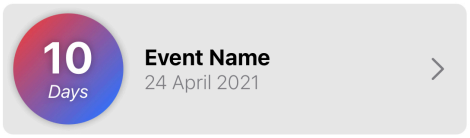
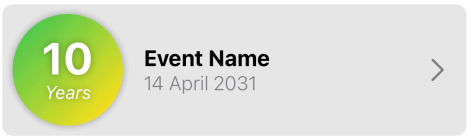
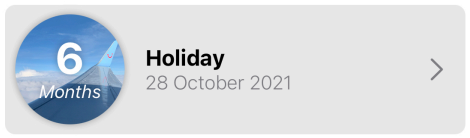
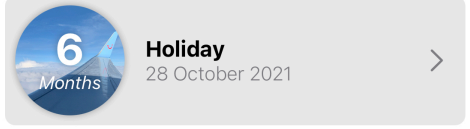
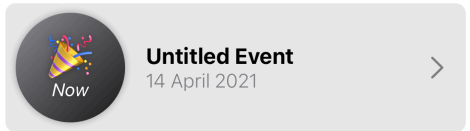
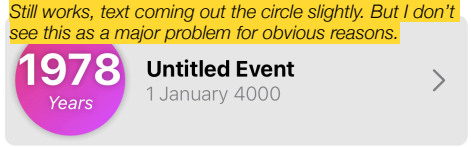
The ‘getTimeline’ function also had to be adapted…

```
func getTimeline(for configuration: ConfigurationIntent, in context: Context,
                 completion: @escaping (Timeline<EventTimelineEntry>) -> Void) {
...

   let widgetTextColor = textColorConvert(for: configuration)

   let entry = EventTimelineEntry(date: Date(), event: eventData, textColor:
                              widgetTextColor, counterType: configuration.CounterType)
...
}
```

The entry that’s now generated now contains the text colour to be used and the counter type to use. In the entry view, I can add modifiers to the text view to change the colour of the text and pass in the counter type to the ‘DateToString’ function to make sure it shows the setting the user selected. That concludes the action I’m taking from feedback.

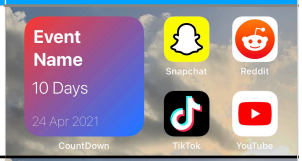
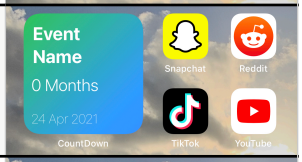
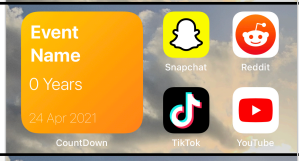
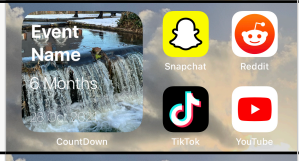
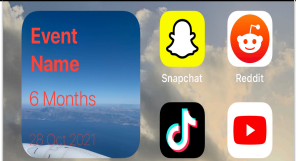
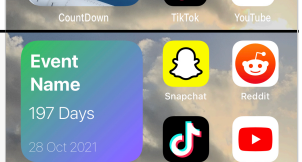
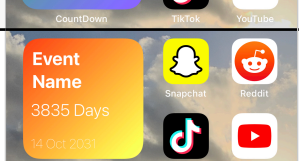
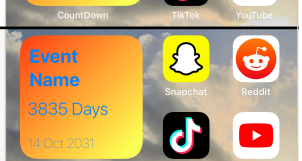
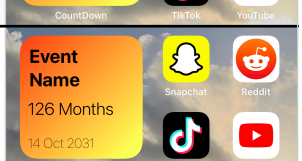
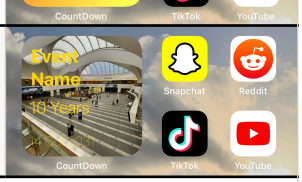
# Post testing & run-though

## *Adding Event*

| Testing / Justification | Test Data | Actual Result |
|---|---|---|
| This is to test the widget creation page. With different sets of data. To make sure the app doesn't crash at any point.<br><br>This is a vital part of the program if you can't create events, that a big problem that will have to be addressed.<br><br>As well as trying with normal data of what I would expect from a user. I will also test with some weird data and there shouldn't be any problems.<br><br>***All of these tests have an expected result of the event added successfully with the relevant countdown and customisation shown.*** | ***Normal Data:***<br>Title: "Event Name"<br>Date: 10 days in future (24/04/21)<br>Colours: Red & Blue<br>Background: N/A |  |
| | ***Normal Data:***<br>Title: "Event Name"<br>Date: 10 years in future (14/04/31)<br>Colours: Green & Yellow<br>Background: N/A |  |
| | ***Normal Data:***<br>Title: "Holiday"<br>Date: 6+ months in future (28/10/21)<br>Colours: N/A<br>Background: *Plane Wing Picture* |  |
| | ***Normal Data:***<br>Title: "Holiday"<br>Date: 6+ months in future (28/10/21)<br>Colours: Blue & Green<br>Background: *Plane Wing Picture* | *Background pic always takes precedent over colours*<br>  |
| | ***Weird Data:***<br>Entering no data, just pressing create event.<br>Testing the default values. |  |
| | ***Weird Data:***<br>Title: "Event Name"<br>Date: 01/01/4000<br>Colours: Pink & Purple<br>Background: N/A | *Still works, text coming out the circle slightly. But I don't see this as a major problem for obvious reasons.*<br>  |
| | There are no ways of causing any error messages to show, and no ways to type in invalid data. For example: A date in the past. Due to the validation I can set on SwiftUI views such as the date picture. | |

***Click me*** - there is a video of me going though the whole app, adding event and widget with comments made throughout. This accompanies the testing tables above and below.

# *Widget System*

| Testing / Justification | Test Data | Actual Result |
|---|---|---|
| This will be testing the widget itself and the different settings.<br><br>Again another vital part of the program, which sets the app apart from other countdown apps on the App Store (at time of release).<br><br>So making sure this works is important to gaining a user base.<br><br>***All of these tests have an expected result of the widget being shown successfully with the relevant countdown and customisation shown.***<br><br>I won't be showing all the combinations but most of them.<br><br>***Customisation options:***<br>- 4 'Counter types'<br>- 10 colours<br>- Background image / gradient<br>- Technically there are at least 11,664 different gradient options, if you only count the grid colours not the custom RGB values.<br><br>So there are technically endless ways to customise it. | Event with colour background.<br>Set to "dynamic" mode.<br>Default text colour (white).<br>Date - 10 days in future (24/04/21) |  |
| | Event with colour background.<br>Set to "months" mode.<br>Default text colour (white).<br>Date - 10 days in future (24/04/21) |  |
| | Event with colour background.<br>Set to "years" mode.<br>Default text colour (white).<br>Date - 10 days in future (24/04/21) |  |
| | Event with image background, no colours also set<br>Set to "dynamic" mode.<br>Default text colour (white).<br>Date - 6+ months in future (28/10/21) |  |
| | Event with image background, colours also set<br>Background image should still take precedent<br>Set to "dynamic" mode.<br>Text colour set to red.<br>Date - 6+ months in future (28/10/21) |  |
| | Event with colour background.<br>Set to "days" mode.<br>Default text colour (white).<br>Date - 6+ months in future (28/10/21) |  |
| | Event with colour background.<br>Set to "days" mode.<br>Default text colour (white).<br>Date - 10 years in future (14/04/31) |  |
| | Event with colour background.<br>Set to "days" mode.<br>Text colour set to blue.<br>Date - 10 years in future (14/04/31) |  |
| | Event with colour background.<br>Set to "months" mode.<br>Text colour set to black.<br>Date - 10 years in future (14/04/31) |  |
| | Event with image background.<br>Set to "dynamic" mode.<br>Text colour set to yellow.<br>Date - 10 years in future (14/04/31) |  |
| | This shows all of the customisation of the widget in action with different backgrounds, dates, colours and 'counter types'. Showing how it all is working exactly how it should. Now obviously I haven't shown all the different combinations (eg: there are 10 colour options). | |

# Evaluation

## Compare to success criteria

***Completed Criteria - CC***

- Widget System
    - Different Colour Text
    - Background Image
- Event Creation
    - Input name and date
    - Input two colours to make gradient
    - Custom background image

Even though there are only 2 points, these are umbrella terms for a very large part of the system.

## Widget System (CC)

After testing it for a few days to make sure that it does tick over every day, I can be confident that it does work. However if you do just sit and watch it at midnight it won't. I assume that's an Apple thing, only allowing widget updates in the background. This is because if went off the home screen for a few minuets and used another app when I looked the widget again it had changed. Further evidence in testing.

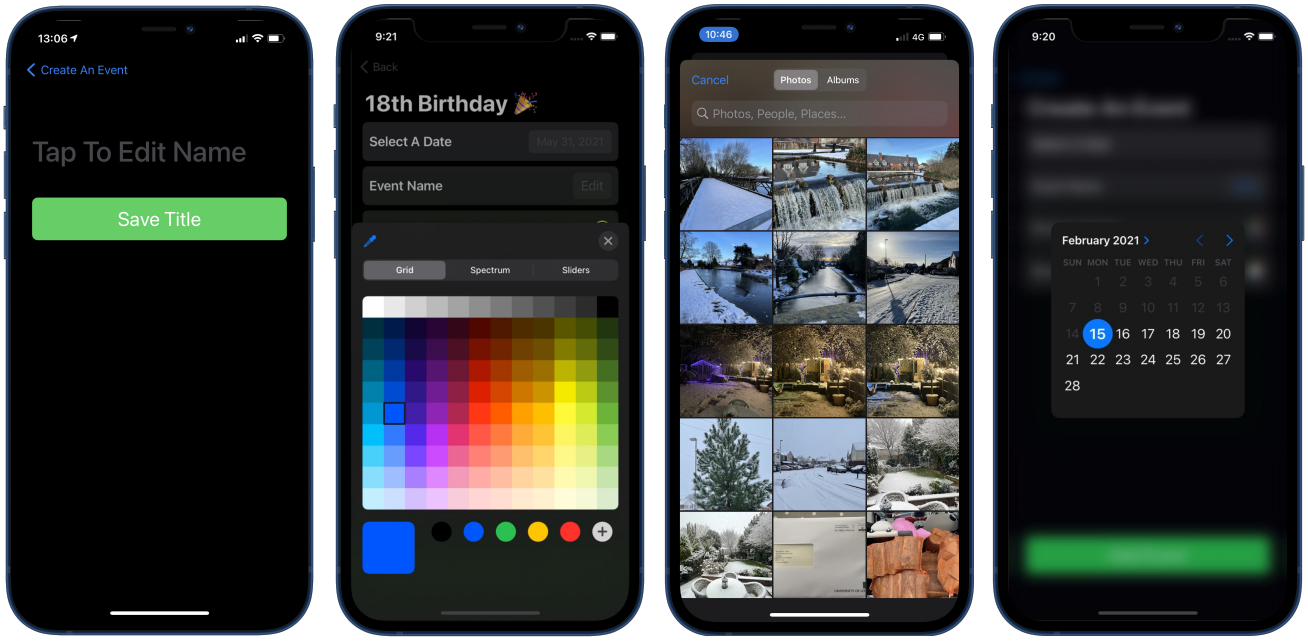
As well as counting down it also has user customisation, which can be done when creating the widget / in the widget settings. All these setting do work and have an effect on the widget. These settings include: different colour gradients, background image, different countdown types and different text colour. A few examples below:

## Event Creation (CC)

Further evidence as well as a video (**link**) is shown above in the post testing section. But as a general overview inputting name and date and its validation (like having a date in the past) is fully functional.

As well as the basics, colour and image selection for images also are fully functional, the only big problem I has was selecting the same colour but as mentioned in development this has been fixed when first testing that section.

Below are screenshots of these features in action:



***What I missed - WIM***

- Organise / Remove widget elements (count, date, name)
- Repeating events (weekly, monthly, yearly)
- iCloud syncing with other devices
- Calendar integration
- iPad version

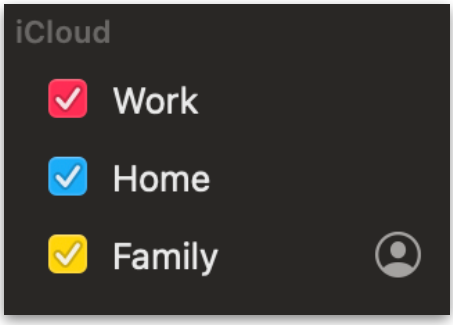
These are the points that missed in my success criteria, there are different reasons for all of them. Going though them in no particular order…

## WIM - Calendar integration

It would of been completely possible to do this feature, using the EventKit framework in iOS. So it wasn't due to limitations of iOS, as long as you have the users permission asking the calendar, creating events and viewing them can be achieved.

One of the main reasons I didn't do this feature was, does the end user even want it. Now I don't know all the needs of the end user. But I thought countdowns automatically created with events on your calendar might be counterproductive, as it would remove that personalised feel of the widget. There has been a very famous study called the IKEA effect, which shows consumers have a greater connection to things them have made. So you could argue autogenerating these would leave the user disconnected from the app. Furthermore not all events in the calendar (in my personal opinion), for example: a countdown to next weeks company Zoom meeting might be more depressing then it is fun, the whole goal of the app.

In addition, if the app also added to the calendar when you create an event it might clutter up someones very important calendar. In the future I could technically add this feature if in a setting page the user could

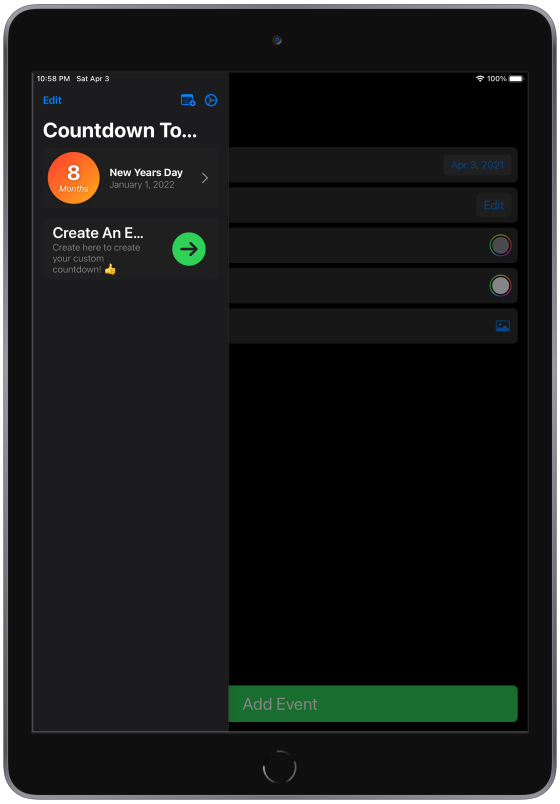


select if this integration is turned on as well as what calendar to use (as you can have multiple calendars on iOS).

Besides my change in hard on the feature since writing the criteria, time was also a factor as obviously I could keep adding more features to this app forever, which could lead to scope creep and make the app more complex then it needs to be. Focusing on improving and streamlining the main features is more important than having a puzzled together app with millions half arsed features.

## WIM - iPad version

Technically there is an "iPad version" because you can download an iPhone only app on an iPad. But that obviously doesn't mean the app isn't optimised for the device, or it easily accessible in the App Store, because obviously optimised apps would be shown first.




All of Apple's devices do use the same programming language (Swift) and code base. Therefore there shouldn't be much tweaking required. Just to prove a point, in the projects settings I allowed it to run on iPad and booted up a simulator. The screenshots below are from an iPad running the app, there have been **NO CHANGES** to the code before running this.

As you can see its fully functioning app, apart from the fact that the main screen is treated as a side bar, which I would have to look into to fix, but overall, you could argue this is partial met because it would only take an afternoon to sort out this niggles in the optimised version. But there isn't a particular rush in doing that due to the probable low download rate, of such a program for iPad.

## WIM - iCloud syncing

This feature slightly links to the previous, as the Apple ecosystem is normally quite tight, you can assume that if a user has one Apple device they are likely to have others. For example: iPad, iPhone, Apple Watch. So if there was an optimised iPad version, the need for iCloud syncing would also have to be integrated, therefore making that simple task mentioned above of just sorting out the niggles of the iPad version would also be followed by the intense task of dealing with this mammoth task.

This is due the fact, the “magic” is expected. Apple users are mollycoddled, which isn’t a bad thing. Thing just connect and work together, so much so it's expected even by 3rd party apps.

Just like how you can text from your phone, Apple Watch, iPad, Mac, HomePod and every other Apple product, users would expect if they had events on their iPhone they could change them on their iPad without the need for manually adding / syncing. The “magic” should still be there.

This feature was 100% not implemented due to time constraints and not understanding how I would share the SQLite database on iCloud, stop conflicts, slow updates, etc.

## WIM - Repeating events

To be fully honest, I thought I would of been quite awkward to add with the current structure of database and would possibly require a complete overhaul late down the development process. Therefore would delay not only the completion of this report and the app but the release on the App Store, which wouldn’t allow me to take advantage of the good timing I had. Which I will get into later.

Now that the finished app is structured like it is, the chances of me doing this feature are now slimmer. As messing with things in production are not a good idea. If you bugger up an update this could

lead to users having problems for weeks as the updates are staggered, and users uninstalling the app.

### WIM - Custom widget elements

Complicated interface and unnecessary. Even though it would be cool to be able to change font, font size, position on the widget, if its even shown, etc.

This would be very complicated, and a lot of questions would need to be ask first:

- Would these be global settings, if you change how the widget looks that applies to all widget?
- Or are the widget settings like font, etc, only related to that specific event, and therefore stored with the event? In the different table?

The interface would be complicated to develop and just like mentioned above with the accessibility issues, giving the user more options and making it more complicated would probably cause more harm than good. Just like what most software development companies have to do, balance catering to the pro users with features such as granule camera control and ease of use. This is one of these cases.

And lastly… like all of them: time.

## Maintenance

No part of this app will need maintenance of any sorts. As this app uses a local database and there is no form of online system, the app could potentially run indefinitely with no issues, counting down to events the user have. Furthermore as mentioned above, when the app is first downloaded a preset event of New Years Day is already there. However this isn't hard coded to a specific year, its coded to always be a year ahead of when the app was launched. For example, if someone downloaded the app in 2025, and event for 01/01/2026, would be created. This demonstrates how no maintenance would be required because I have already taken it into account.

The app would only work in the semi-long term, because obviously if it was left decades (which is an extreme example), assuming that the phone was kept up to date, eventually it wouldn't due to compatibility issues. Just like how 32-bit applications will not run on the latest macOS version, only 64-bit. Meaning that developers would have to update their app if they haven't done so already.

However by using SwiftUI which is a very new framework only being released to the public in late 2019. It is very new compared to its older counterpart FoundationKit and UIKit which was opened up to developers alongside the release of iOS 2.0 in 2008. Therefore as I'm using the newest frameworks I wont have to worry about mass depreciation unlike a program using UIKit, which might be fully deprecated by Apple in the next few years.

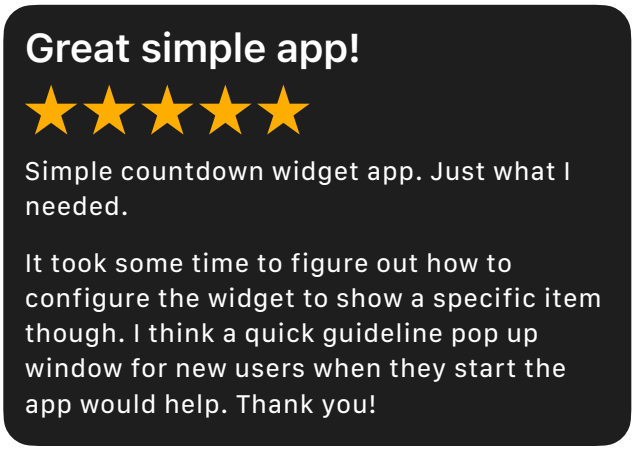
But as mentioned, if I left it now and never updated the software again it would be fully functional for many years.

Now if someone took over development of the software, the comments already in the code should be able to create a basic understand. Furthermore due to my choice of putting key functions which relate to each other into separate files, for example: 'DatabaseConnector', they can be easily referred to in code as well as having the two files open side by side to work out how certain sections will work.

## Limitations

### Usability

As mentioned in the email above, and other users in general. A few users have struggled finding and editing the widget setting. Even though this doesn't actually link to the app fully, as you could argue this is a problem with Apple's



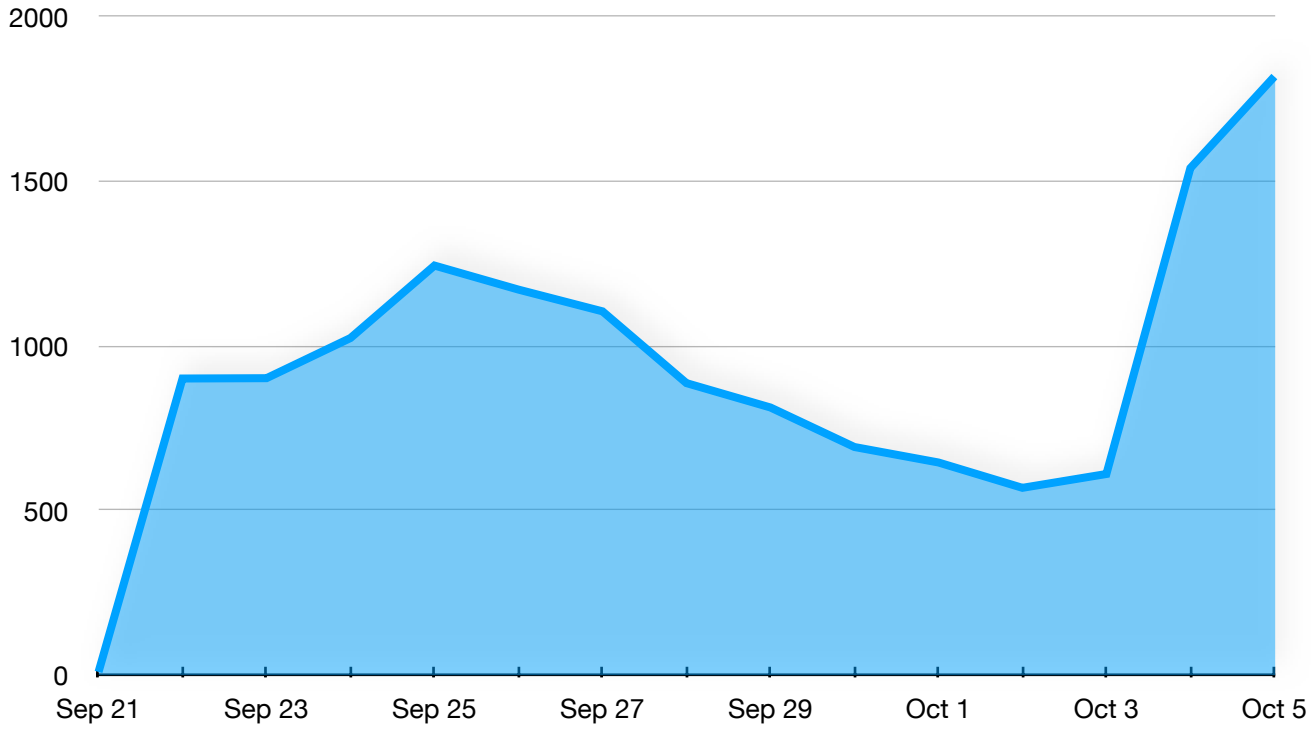
usability in general. All of these users suggested to add a little information screen or video to the app to explain how to do this. Usability is all about guiding your user though your application, either by explicit means such as help boxes or just in the way it's designed, leading to them understand how the application works. In this case I did none, meaning the user had to struggle to work it out, or even contact me directly.

Usability is forced by Apple, if you are making an App for the App Store then it needs to follow the **Human Interface Guidelines**. Everything from how the user interacts with the app, how its designed, event flows, etc. All have already been predefined. Now I don't have time to go over what I've applied in detail. An example of a guideline is: "**Provide ample touch targets for interactive elements.** Try to maintain a minimum tappable area of 44pt x 44pt for all controls". So due to these strict rules this means my app already meets a very high standard, the consistency between apps the Apple have created means that users can look at an interface and know how to interact with it, just from prior experience.

## Releasing the app on the App Store

As I am a fully licensed Apple Developer that gives me the ability to publish an app on the App Store. Now as mentioned, with this app I published it in perfect timing just after the release of iOS 14 meaning there weren't many apps on the App Store with countdown widgets. Furthermore just like with any big software Apple, there was a lot of news coverage of the new features. On top of this was a TikTok trend based around using this freedom to customise your homepage with widgets, etc. The combination of these things, lead to staggering statistics after releasing my app.

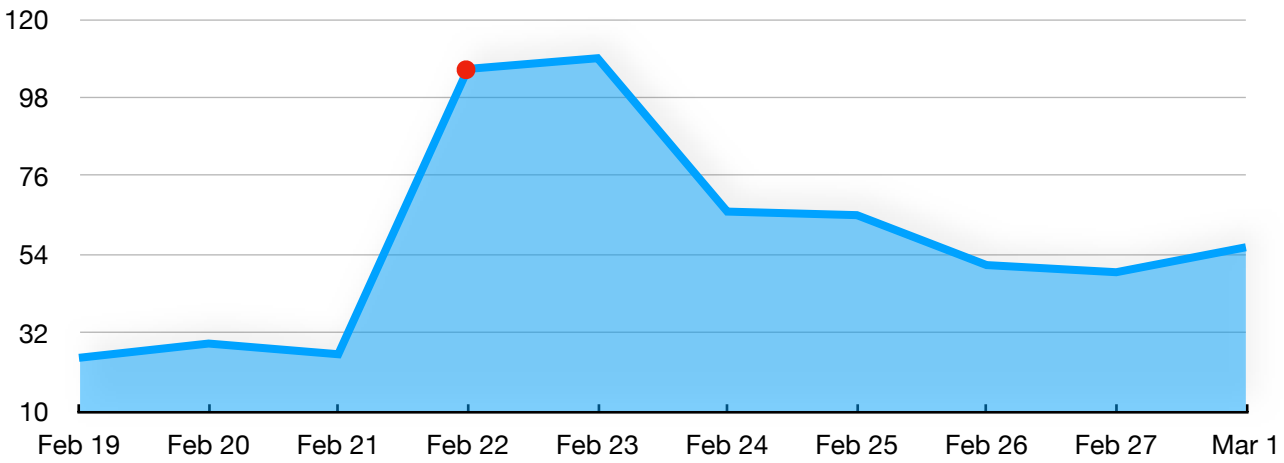
Within 2 weeks of release, I had ***14,000*** downloads…




After a large spike right at the beginning of release, downloads started to drop and level off at around 600 a day for a few months. Then from the start of 2021 onwards around 300 a day on average, with random spikes in-between.

## An interesting spike

Now there was a few interesting spikes over the past few months, but this was the most interesting. If you look at **downloads only from the United Kingdom** between the 21st and 22nd of February, there was an increase of **308%**, due to people counting down stages of the COVID roadmap, announced by Boris Johnson on the 22nd of February 2021.



## Reviews

Due to the large amount of downloads, comes a large amount of feedback, here are some of the reviews I've received from all over the world.

### Just yesss, download ASAP

★★★★★

Thank you sooo much for making such an amazing app and making it all *FREE*!!! The widgets truly work and amazingly, being able to customise the text and all! It took me a good 15 minutes downloading and deleting all the other countdown apps as they asked money and weren't customisable enough, but this- THIS takes the cake tbh...

I 100% recommend this app to anyone reading this and I will also be recommending this to my family members!

Note to the creator: tysm!!!! Honestly the work is amazing! And I couldn't ask for more! Keep doing what your doing it's truly amazing!!!

### Short feedback

★★★☆☆

Thanks for the app! Just tried it out some minutes. Short feedback; would be nice being able to "disable"/remove a picture and just use color background. As per now if I choose a pic I'm stuck on the opt to use a pic. Best J

### Just Works!

★★★★★

I have tried a handful of countdown apps that claim to be widgets but have been far too difficult to figure out or is not actually a widget for your Home Screen. Glad I gave this one a go because it is simple to use and does exactly what it says it does!

*And loads more from around the world...*

### Amazing app

★★★★★

Does what it is suppose to do. Helps us keep track of time and the events that pass in a beautiful way.

### Very nice countdown widget

★★★★☆

So far, I love this app. I tested out basically every app i could find that advertises having a widget and so far this is the most reliable, clean and easy to use. Initially there were some bugs with the app, but they were quickly patched out within a few days after the launch of iOS 14 so it is nice to see that it is being tended to.

### Best countdown app!

★★★★★

This is the best countdown app I've found! It isn't cluttered with ads and you don't have to pay extra for the widgets.

### Суперское

★★★★★

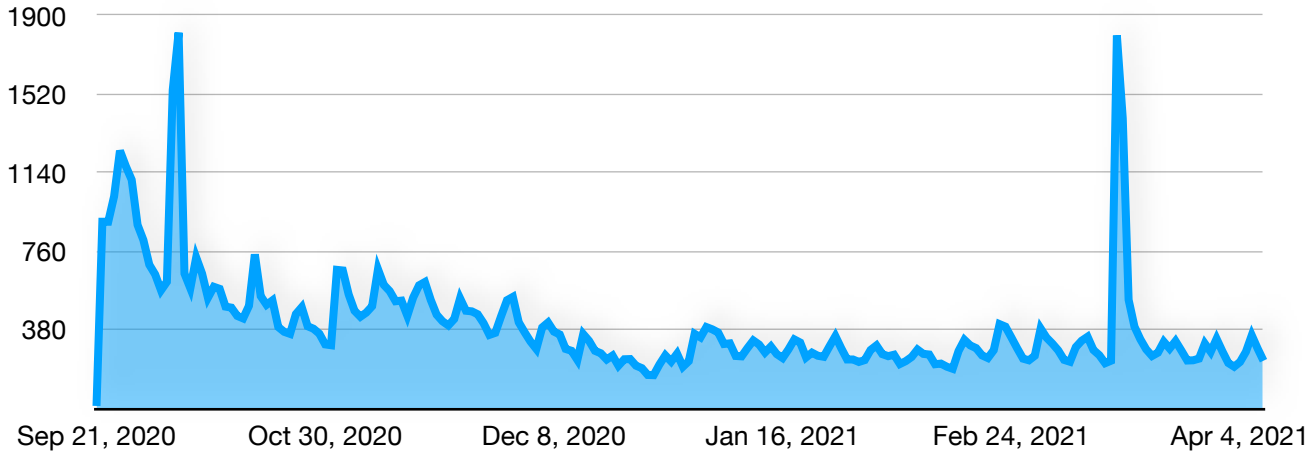
Давно искала такое приложение.
Суперское, очень нравится

***Translation:***
I have been looking for such an application for a long time.
Superb, really like it!

### Parfait

★★★★★

Super app pour faire des compte à rebours, je recommande ! 👌🏽




## Final note

Overall my app has been **downloaded 90,000 times** in just over six months. Which I would say is one hell of an achievement for a one man developer. This really shows how with the power of technology we all have the ability to have an impact. My code will be executed on tens of thousands of devices every day. In **over 100 territories** worldwide.

So simply, my app works.

# Comments

## A few random final comments

Here is just some random comments about my app at time of writing.

- The version of the app on the App Store currently (V2.9) is not the same to how the app was during writing of my development.
- There are more features in version 2.9 of my app. Such as:
  - **Settings page:** which addressed the limitation of users not understanding how to add a widget.
  - **Settings page:** A button to "Buy me a coffee" as a form of donation.
  - **Settings page:** Importing and exporting the SQLite database to the files app, this is actually part of my success criteria. But at this point that's just more to write.
  - **Settings page:** Button to email me, which is why I have so many emails to reference in this project.
  - **Add event page:** It has just been updated to be more user friendly as well as better for smaller devices.
  - **Widget:** You can now display your countdown in binary because why not.

I also have more featured planed for the future such as an Apple Watch extension to view your countdowns on the watch face. As well as a medium and large size widget to display multiple countdowns at once. This is planned to also be an "In App Purchase" as a way to monetise the app.

It's actually been quite interesting to write (however some of my peers disagree on that). Writing the development section was especially interning as a way of looking back though some code at points and even having to research frameworks such as "WidgetKit" and "App Groups" in more detail so I was able to explain it in detail in this report.