2017

# IOS CALENDAR WEEK VIEW

EVAN COOPER

| | |
|---|---|
| DATE: | DECEMBER 15TH 2017 |
| SUPERVISOR: | LOUIS D. NEL |
| DEPARTMENT: | SCHOOL OF COMPUTER SCIENCE |
| ORGANIZATION: | CARLETON UNIVERSITY |
| COURSE: | COMP 4905 – HONOURS PROJECT |

## ABSTRACT

Mobile application Integrated Development Environments (IDEs) provide a wide range of pre-made libraries that aid developers in building applications for mobile platforms. Xcode, Apple's IDE for developing iOS applications for iPhones, iPads and more, is no exception. Their extensive repository of user interface (UI) elements allows developers to focus on the core functionality of their application, rather than building an interface from scratch. Alternatively, third-party sources such as CocoaPods, host even more (over 37 thousand) (CocoaPods Dev Team, 2017) open-source libraries to alleviate the limitations of Xcode's default widgets. However, there lacks one library that can be very useful to a wide range of applications; the calendar library with a week-view interface.

This project has successfully created an original library, written completely in iOS' native language: Swift. It's graphical and programmatic interfaces mimic that of a native library included in Xcode. The high-level data source and delegate protocols provide simplistic access to the complex functionality under the hood. Modular design offers a wide range of customization and usage to the varying protocols. Required data can be loaded synchronously or asynchronously without complex code, and interface elements can easily be changed at individual levels.

## ACKNOWLEDGEMENTS

### *SWIFTDATE*

Date and Time Management is provided by the SwiftDate library.
Available at http://www.swift-date.com
Created by Daniele Margutti and licensed under MIT License.

SwiftDate was used to alleviate the overhead required to create sophisticated date objects. The use of this dependency is to have consistent time parameters for event object.

## TABLE OF CONTENTS

## LIST OF FIGURES

## MOTIVATION

I first became aware of the need for this type of library while building another mobile application for a project during my third year of undergraduate studies. Throughout the semester, we were required to build a mobile application to completion, on a chosen platform, in small groups of students. Schedula, our application, had a main goal to automate the course selection process for Carleton University students by generating all permutations of class schedules that would work for each student on an individual level. The idea for this application arose when my group and I were reflecting on the amount of manual effort required to register for classes that would fit well with our non-academic lives, such as work and other time commitments. When building a schedule with our application students have the ability to opt-out of entire days of classes. They can also indicate a preference for morning, afternoon or evening classes, and even input specific blocks of time during any day where they wouldn't be available to attend classes.

After designing and implementing the software to achieve these goals, the next step was to present it to the user. The view that displayed the data to the user needed to have two primary properties: concurrently display all five weekdays at the same time, and the ability to see the different classes occurring each day, similar to Carleton University's mobile application that a student uses to view the classes they have for the week [Fig. 1]. However, neither iOS or Android development environments have built-in user-interface widgets to display data in this manner. After extensive research, a third-party library for Android was found on GitHub, that satisfied the needs of the interface. This library, called Android Week View, is sophisticated and simplistic. This forced the application to be developed for Android devices, and still left the domain of Apple's iOS untouchable. However, it allowed the development cycle of the class project to focus on the core functionality, by alleviating the need to develop an intricate interface element from scratch.

A year later, following the completion of the class project, iOS still didn't have a native or third-party library that fulfilled the needs of my third-year course scheduling application, or of those who used Android Week View in their own applications. I started to wonder if I could develop my own week view interface myself, but was uncertain as to why it had not been created already. My main justification was that developers thought the effort required would outweigh the benefit, since Swift has a history of drastically changing, due to its open source language and the fact that it is relatively young when compared to the other popular languages used today. Nonetheless, Swift's release cycle has been slowing down over the last year, as it approaches its endgame releases, and the impact of this reasoning is becoming less significant (Apple, 2017). Other rationales included that I have yet to find someone else's version of a week view, due to poor search queries on my side, incomplete SEO's from the developer, and even the pure size and diversity of the internet. However, I didn't believe this to be the case, as the difference in current search results and those from the previous year were minimal. The idea that I could develop my own version for iOS platforms became feasible.

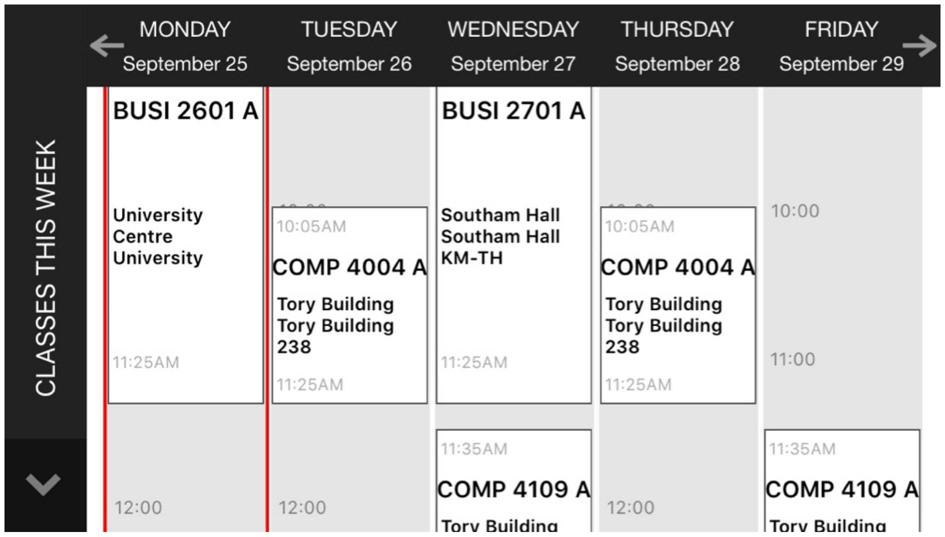


FIGURE 1 – Carleton University's mobile application's "Weekly Schedule" section.

## METHODOLOGY

### INTRODUCTION

The primary purpose of this project was to design an iOS library that can be used by developers within their mobile applications, no matter the context of their application/project. Moreover, this software is not meant to be used by the general consumers of iOS products, and is not a full-fledged app its self. Throughout development, extra care was taken into place to ensure a modular design with utmost compatibility. Hence, this library and its single dependency is also written completely in Swift. Not only is this project written in a single language, it also attempts to mimic Swift design guidelines, so that it feels native to the language and environment.

In iOS development, there are two available languages, Objective-C and Swift. Although Objective-C is a language with more history and experienced users, there are many other reasons to why Apple's Swift language was chosen for development. Primarily, it is the new language of choice by Apple developers, and is the main language used in modern iOS, macOS and tvOS Software Development Kits (SDKs). Since it is built on-top of its predecessor and the C programming language, it inherits many of Objective-C's abilities, while at the same time adding its own advantages. Readability in a programming language is crucial, especially when building libraries that are open-source. Since the goal of this project is to develop an open-source iOS library, the source-code needs to be easily readable to the developers who may use it, to streamline integration and minimize errors. Not only is it more readable than Objective-C, its resemblance to other programming languages like JavaScript and Python makes understanding even easier to today's developers.

There are two primary and distinct classes that drive the performance of this library. The first is the UIInfiniteScrollView class, which subclasses and builds on top of Swift's UIScrollView class. The main goal of this class is to add and handle certain scrolling abilities that are not present in UIScrollView. Secondly, WeekView is the class that is meant to be used by users of this library. It implements UIInfiniteScrollView and adds any functionality related to calendars.

## UIINFINITESCROLLVIEW

### DESCRIPTION

The first step to building a comprehensive UI library, was to develop a sub-system in charge handing some core low-end functionality that would stay consistent throughout any and all iterations of an iOS Calendar Week View. Primarily, this library needed to display a large set of views with the same structure known as sub-views, each representing a single day. Additionally, it was necessary that this layer could handle the scrolling functionality, and provide the fluidity of an horizontally-infinite space, so users can freely scroll between each day. Initially, Swift's UICollectionView and UIScrollView classes were promising candidates to fill both requirements of the sub-system. Being "[A]n object that manages an ordered collection of data items and presents them using customizable layouts", the UICollectionView already has the proper structure and interface to be filled with a view that represents each day (Apple, 2017). However, the UIScrollView class gives more freedom in terms of "scrolling and zooming of it's contained views" (Apple, 2017). Both of these interfaces quickly became attractive options as they had been developed by Apple themselves, and are robust and sophisticated libraries. However, these views lacked a specific ability that deemed them invaluable to the project: on a horizontal plane, they were only able to extend the plane on the right side, and thus created a left-hand scroll barrier and rendered themselves useless on that side. Ultimately, this meant that developing a custom interface for the core functionalities would be necessary.

The custom interface that successfully handles the underlying functionalities is known in this project as the UIInfiniteScrollView class. An instance of this class has two core abilities: horizontally or vertically infinite scrolling in both directions of the chosen scroll plane, and provides higher level classes a Swift-style interface for continuously initializing the views that would have a similar structure. To be able to keep track of the scrolling activities of a user, UIInfiniteScrollView subclasses UIScrollView, hence the similarity in names. Apple's pre-built library includes various scroll listener functions that are called during specific events. To take advantage of this powerful functionality, this interface overrides a couple key functions that, together, allow for the constant tracking of scroll position. This ability was fundamental to providing an experience of an infinitely long plane.

The table below describes the different parameters within the UIInfiniteScrollView class. Although most of the parameters within the class have a private access level, understanding the importance and use of each parameter is critical to being able to properly use this library.

| Parameter Name | Type | Description |
|---|---|---|
| views | [[UIView]] | A nested collection where each index at the top level is a collection of views meant for a specific row/column in the scroll view. |
| viewRangeStart | Int | An integer that represents the index at which to start loading the content of each row/column from "views". |
| loadPageCount | Int | An integer representing the number of pages that are active within the scroll view. |
| viewSize | CGSize | The size of each row/column in the scroll view. |
| spacerSize | CGFloat | A float representing the amount of spacing between each row/column. |
| viewsInPageCount | Int | An integer representing the number of rows/columns within each page. |
| scrollDirection | ScrollDirection | The direction in which the scroll view scrolls. Either vertical or horizontal. |
| isSnapEnabled | Bool | A Boolean that indicates if the scroll view should snap the closest row/column to the edge after a user has scrolled. |
| weekView | WeekView | The WeekView instance that initializes this class. |
| dataSource | UIInfiniteScrollViewDataSource | An object that implements the UIInfiniteScrollViewDataSource protocol. |

## SCROLLING

Infinite scrolling, either in the horizontal or vertical directions, quickly became one of the most difficult challenges to overcome during the course of this project. When using Xcode's pre-packaged interface packages on a horizontal plane, infinitely scrolling to the right only required extending the width of the plane by some arbitrary size that allowed the addition of a couple more views. On the contrary, infinitely scrolling to the left was not as simplistic. Adding space to the leading edge of a view requires the entire view to be re-initialized, and all of its contents to be re-added. The same concepts applied to vertical scrolling planes, where adding content to the bottom is simple, but not for top content. Analysing the work being done during this process revealed a linear relationship between the re-initialization time of the interface and the amount of times the interface has already been re-initialized. This meant that adding a single leading view to the scrolling plane would additionally re-load every view that existed before it. Since all user interface activity within iOS applications must execute on the device's main thread, handling user interaction and infinite scrolling in this manner would quickly lead to poor performance (Apple, 2017).

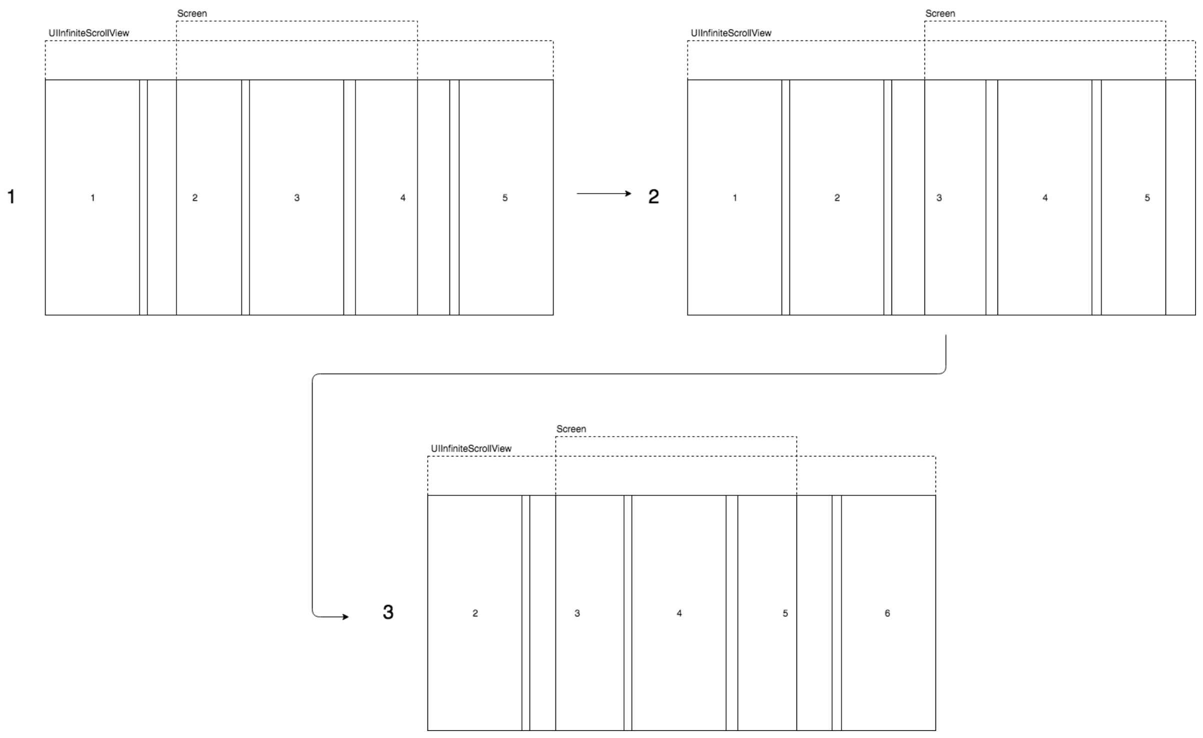
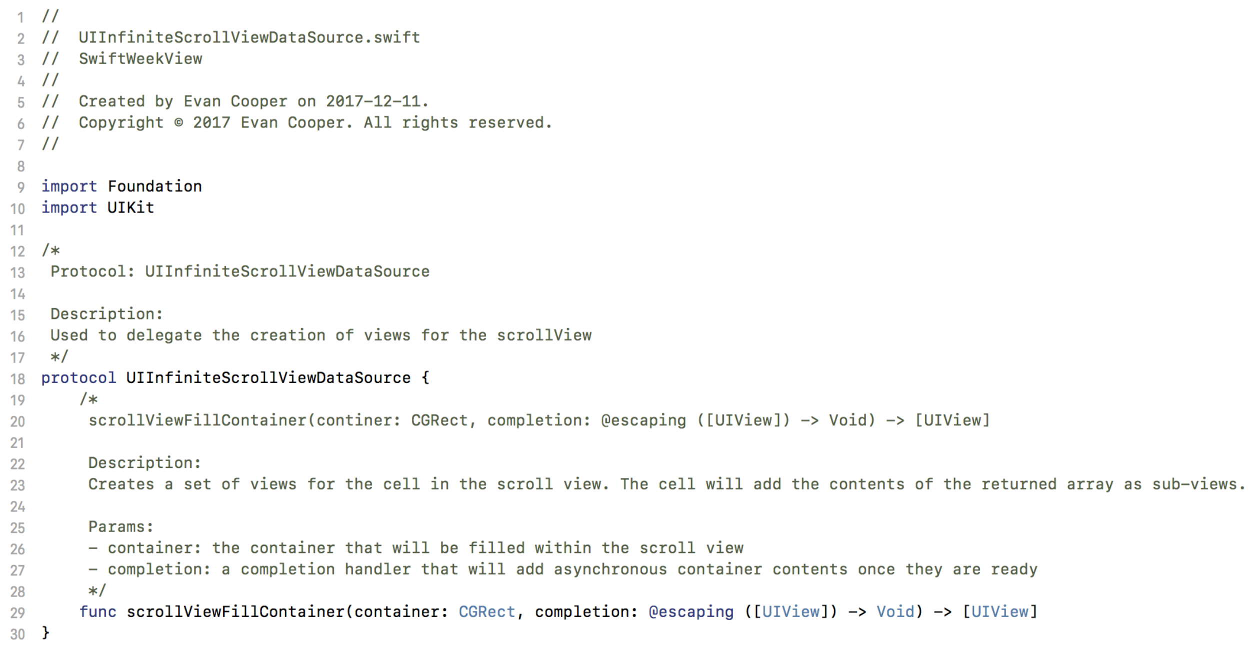


FIGURE 2 – UIInfiniteScrollView scrolling process

In order to avoid a linear relationship in UI initialization that could freeze the application as a whole, a constant amount of real-estate on the scroll plane needed to be loaded each time. The solution was to only load in views that are relative to the current position of the screen. These views fill the available space of available in the scroll plane, and the screen can freely scroll through them. To provide the appearance and experience of an infinitely scrolling view, UIInfiniteScrollView implements and uses the delegate methods of Swift's UIScrollView to act on specific user-initiated events. When the visible content within the scroll view is detected to be nearing either edge of the loaded content, the UIInfiniteScrollView will re-initialize the content all the content within, and re-position the screen to the middle of the content. This functionality makes it appear as though there is no end to the scroll view's content, since the user's screen will never reach the end, and they will not feel resistance when scrolling the screen [Fig. 2].

## ACCESS TO FEATURES

For high-level classes to utilize this framework, UIInfiniteScrollView provides an easy to use interface in the form of a Swift-style protocol. Swift protocols define "blueprint[s] of methods, properties, and other requirements that suit a particular task or piece of functionality" (Apple, 2017). The advantage to using protocols is that certain functionalities and tasks for a given class can be delegated to other classes, as long as they properly conform to the protocol in question. In Swift, UI objects often have two primary protocols; the data-source and the delegate. Objects that act as a data-source will often provide the data that will populate the class that makes use of the data-source. Delegate objects often conform to the observer pattern (a term made mainstream by the Gang of Four) by being able to receive events and actions, and then perform operations when triggered by the primary class (Gamma, Helm, Johnson, & Vlissides, 1994). In the context of this project, since the only events that will be triggered by UIInfiniteScrollView are related to scrolling, and they will always be handled the same, only a single protocol has been implemented. UIInfiniteScrollViewDataSource protocol [Fig. 3] defines a single method with the task of creating each row or column as its needed, at runtime. The protocol method is called synchronously on the main thread, with an additional completion handler for other usage scenarios. In order to properly take advantage of this protocol method, a skeleton for the row or column should be immediately returned in synchronous fashion, while additional components that either need to be retrieved from the internet or simply take longer to create in general should be added via the completion handler, so not to disrupt the main thread. It is important to note that objects passed to the completion handler will be added as sub-views to the row or column that is being created. For example, in context of a week-view calendar, the static structure of the calendar's day should be returned to the protocol method, while events for the day should be passed to the completion handler.



```swift
//
//  UIInfiniteScrollViewDataSource.swift
//  SwiftWeekView
//
//  Created by Evan Cooper on 2017-12-11.
//  Copyright © 2017 Evan Cooper. All rights reserved.
//

import Foundation
import UIKit

/*
 Protocol: UIInfiniteScrollViewDataSource

 Description:
 Used to delegate the creation of views for the scrollView
 */
protocol UIInfiniteScrollViewDataSource {
    /*
     scrollViewFillContainer(continer: CGRect, completion: @escaping ([UIView]) -> Void) -> [UIView]

     Description:
     Creates a set of views for the cell in the scroll view. The cell will add the contents of the returned array as sub-views.

     Params:
     - container: the container that will be filled within the scroll view
     - completion: a completion handler that will add asynchronous container contents once they are ready
     */
    func scrollViewFillContainer(container: CGRect, completion: @escaping ([UIView]) -> Void) -> [UIView]
}
```

FIGURE 3 - UIInfiniteScrollViewDataSouce definition.

## ADDITIONAL ISSUES & SOLUTIONS

The improved scroll-handling method, although allowing for a constant relationship in UI initialization, has a trade off in terms of data storage. Given the fact that old, unused sub-views within UIInfiniteScrollView will be removed to make way for the new ones, these views that are being removed must be saved to disk, to allow for them to be re-loaded when the user scrolls back to its position on the scroll plane. Additional problems also arose, including saving these views to disk using a data structure that facilitated easy retrieval of specific elements, and re-adding views with extreme coordinates that lie outside UIInfiniteScrollView's content-view scope.

Despite the fact that this one-to-one relationship between disk storage and rows or columns created has emerged from the improved scroll mechanism, it has far less of a negative impact on performance than the original scrolling system, as the same data that is being saved to disk would instead to be saved to memory. In addition to the main thread potentially being locked, storing large amounts of data in system memory will eventually become detrimental to the performance of a mobile application and the device itself. This would have occured especially quickly when working with data types that have significantly large sizes, such as videos or images. In order to re-load these sub-views when they're needed, a couple additional steps were required in order to alleviate new issues that arose. Firstly, the data structure that saved all the views needed to be arranged in a certain way that would make them easy to retrieve. The easiest method was to save the views in an array, and sort the array as the views appear on screen; by their origin's x or y value as it would appear on an infinitely large plane, depending on whether UIInfiniteScrollView scrolls horizontally or vertically, respectably. The primary issue that was consequence of retrieving views from disk was re-adding views that extended past the content view of UIInfiniteScrollView. For example, views that live on the extremities of the scrolling plane, have coordinates that are either too far left or right of the content view, as it has a limited width, with an origin that always has x and y values of 0. When these edge-views were added to the content view of UIInfiniteScrollView, they were never visible to the user since the screen would constantly reset when nearing the edge, as design of the scroll mechanism.

Combatting this issue proved to be more difficult as new features were added to the project. It was evident that the views' origin coordinate needed to be modified so that it would fit within the screen, however, the modification of the coordinate would remove any order that was already established in the array that held these views. The solution was to make a copy of the view that should be added, and add the new instance to UIInfiniteScrollView. The coordinate modification proved to be a simple task, since only one of the x or y values of the copied view needed to be modified, and the other would stay constant at zero. However, when a copy of a view was made a larger problem arose. When copying these views, there were often attributes that were lost in the conversion, and some cases, copying subclasses of Swift's native UIView returned an object of the UIView class and not the original subclass. The most important attribute that was often lost was the array of gesture recognizers, which made it impossible to interact with most views that were added to UIInfiniteScrollView. The solution is a two-step process. First, any UIView subclass properties needed to be explicitly encoded and decoded by implementing Swift's encode function and decoding initializer. The second step goes against object-oriented programming's practices, by forcing the copy of the original view to be a class of the created subclass. This step is not good programming practice, as it breaks a barrier between the UIInfiniteScrollView class, and the class that implements the delegate by hard-coding and enforcing a variable to be of a certain class, when that class lacks any relationship with UIInfiniteScrollView.

## WEEK VIEW

### DESCRIPTION

The final step towards completion of this project was to develop a complete interface on top of the completed foundation. Since many parts are already being handled in UIInfiniteScrollView, the WeekView class only had to focus on specific properties and functionalities directly related to calendars. Primarily, this class needed to create new calendar events for each day and display them.

Initialization of a WeekView, there are many different processes that take place. Starting initialization can happen in different ways and can be achieved with a very small set of user-provided properties, as initializing functions define default values for almost all class-attributes.

If an instance is being created within code, only the frame of the calendar and the count of days that are concurrently visible are required. All the other properties have default values and can be optionally passed into the initializer. Storyboard initialization is even easier, as the calendar's frame is defined explicitly within the layout files, and the visible days property takes on a default value and can be optionally modified within code after establishing a connection between the WeekView in the storyboard files as an outlet within the relevant view controller.

After initialization of the primary properties that were either provided by default or the user, the calendar will build a skeleton for its own interface. Most importantly, it will create an instance of UIInfiniteScrollView, and define its self as the scroll view's delegate since it conforms to the UIInfiniteScrollViewDataSource protocol. The scroll view will display all the It will then build the time view, which is visible on the left side. This view displays a selected or default range (09:00 to 17:00) of hours during the day, which is used to easily distinguish the time at any point within the calendar. Lastly, a top-anchored view is created which displays the month and year of the calendar.

The table below describes the different parameters within the WeekView class. Although most of the parameters within the class have a private access level, understanding the importance and use of each parameter is critical to being able to properly use this library.

| Parameter Name | Type | Description |
|---|---|---|
| monthAndYearText | UITextView | A text view that displays the month and the year. It is at the top of the WeekView. |
| timeView | UIView | A view that displays each hour of the day, or specified time interval. It is on the left of the WeekView, under monthAndYearText. |
| scrollView | UIInfiniteScrollView | The scroll view that will display each day in the calendar, along with the events for that day. It is to the right of timeView and under monthAndYearText. |

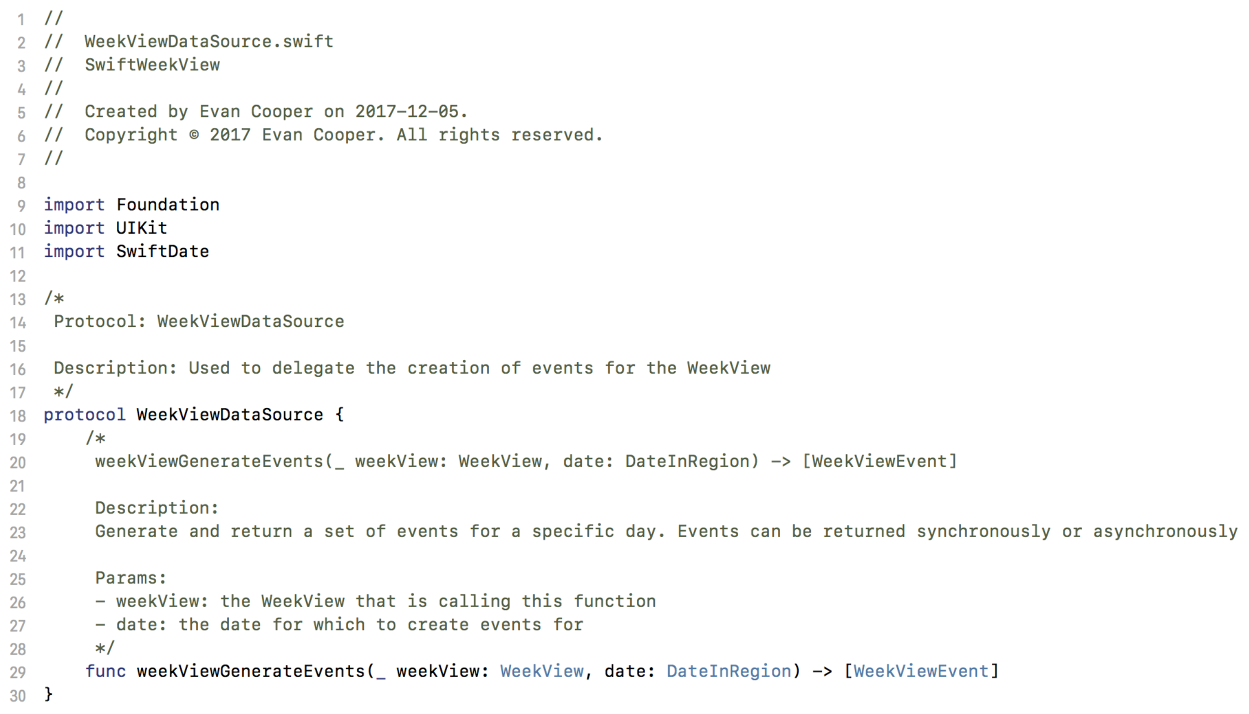
| | | |
|---|---|---|
| events | [WeekViewEvent] | A collection of WeekViewEvents that were created WeekViewDataSource protocol. |
| initDate | DateInRegion | The initial date that the WeekView will open at. |
| visibleDays | Int | The amount of days that are concurrently visible. |
| startHour | Int | The first hour of the day to display within the WeekView. |
| endHour | Int | The last hour of the day to display within the WeekView. |
| headerHeight | CGFloat | The height of the header included in each day, displaying information such as the day of the week and day of the month. |
| respondsToInteraction | Bool | A Boolean value indicating if the WeekView should respond to user interaction with individual events. |
| nowLineEnabled | Bool | A Boolean value indicating if the WeekView should display a line a the current time. |
| colorTheme | Theme | A theme used for coloring the components of the WeekView. Either light or dark. |
| font | UIFont | The font used throughout the WeekView. |
| nowLineColor | UIColor | The color of a line that displays the current time in WeekView |
| nowLine | CAShapeLayer | The line and the circle that, combined, display the current time in WeekView. |
| nowCircle | UIView | |
| dataSource | WeekViewDataSource | The object that implements the WeekViewDataSource protocol. |
| delegate | WeekViewDelegate | The object that implements the WeekViewDelegate protocol. |

| | | |
|---|---|---|
| styler | WeekViewStyler | The object that implements the WeekViewStyler protocol. |

## WEEK VIEW EVENTS

Displaying calendar events is the core objective to this project. Since they are so fundamental to the design and flow of the library, the creation of the WeekViewEvent class was necessary. This class represents a calendar event that would be typically displayed in a calendar. Most importantly, it houses *start* and *end* properties that together represent a specific interval in time to which the event occupies. Additionally, it the event has a *title* property that is meant for a clear and concise description of the meaning of the event. Unimportant to a user of this library, but fundamental to internal logic, WeekViewEvents self-initialize a unique identifier (UUID) that easily distinguishes its self from all other calendar events.

Adding events to the calendar makes use of WeekView's data source protocol method, WeekViewDataSource [Fig. 4]. Conforming to this protocol is the most important step to using this library, as it is the single access point to populating a calendar with events. For each day, this protocol's only method, weekViewGenerateEvents, is called and expects a collection of events that occur during that day. Each time that the protocol method is called, it provides the implementing class with the date of which the events are expected to occur within. In a calendar, it makes sense to separate events by the day that they occur, as most only last for a few hours a day, and rarely span across multiple days. Unfortunately, due to this implementation, calendar events that do span multiple days are sometimes incorrectly represented to the user by only being visible in the first day that they occur. A work-around solution for this issue is to separate the event to a day-by-day basis, and return each day individually to the protocol method.



```swift
//
//  WeekViewDataSource.swift
//  SwiftWeekView
//
//  Created by Evan Cooper on 2017-12-05.
//  Copyright © 2017 Evan Cooper. All rights reserved.
//

import Foundation
import UIKit
import SwiftDate

/*
 Protocol: WeekViewDataSource

 Description: Used to delegate the creation of events for the WeekView
 */
protocol WeekViewDataSource {
    /*
     weekViewGenerateEvents(_ weekView: WeekView, date: DateInRegion) -> [WeekViewEvent]

     Description:
     Generate and return a set of events for a specific day. Events can be returned synchronously or asynchronously

     Params:
     - weekView: the WeekView that is calling this function
     - date: the date for which to create events for
     */
    func weekViewGenerateEvents(_ weekView: WeekView, date: DateInRegion) -> [WeekViewEvent]
}
```

FIGURE 4 - WeekViewDataSource protocol definition.

Not only is the WeekViewDataSource the most important protocol for WeekView, it also has the most complex functionality. When the WeekView needs to call upon the data source method, it will start a new background thread to retrieve the collection of events, meaning that weekViewGenerateEvents is actually called asynchronously. In a developer's perspective who would use this library, this implementation allows for their events to be fetched synchronously or asynchronously from the internet, without any additional overhead. In Swift, threads are started and managed by the DispatchQueue class, which is simply a pool of tasks. Tasks are allocated CPU time by the iOS device its self, based on the type of operations that will be completed. For example, every iOS application has a high-priority user interface thread (UI) who's main purpose is to make changes to the screen that a user sees. For the WeekView, each task added to the pool is a single call to the data source protocol for a single day. The challenge that occurs when working with threads in any programming language and system, is that there is no guarantee provided by the computer that one thread will finish before the other. Solving this issue, and having the ability for out-of-order execution is known in computer science as concurrency. The WeekView is able to compute events concurrently by executing completion handler method on the main thread after the events have completed. The completion handler uses the collection of generated events to create the views to represent them, and send them to its UIInfiniteScrollView's data source protocol completion handler to be added.

Comparing calendar events is equally important as creating them. For example, it is important to determine whether two calendar events overlap with each other, so that they can be displayed properly. Without the ability to compare the timing of calendar events, users may not be able to see or interact with certain events, if they appear directly behind another. Implementing Swift's Comparable protocol, by defining the "less-than" and "equal-to" operators allows for simplistic determination on which event occurs before another. By comparing event timelines against each other, events that occur at the same time can be displayed properly. Showing events that occur at the same time requires their width to be divided proportionally, and their x-coordinate be modified accordingly [Fig. 5].

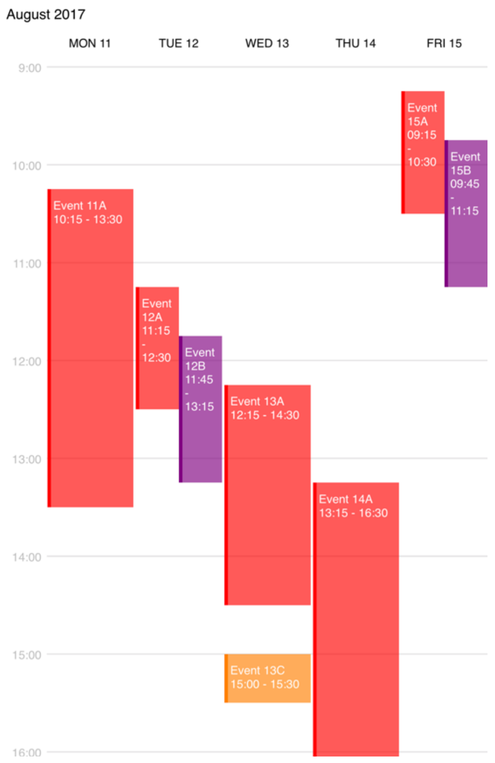
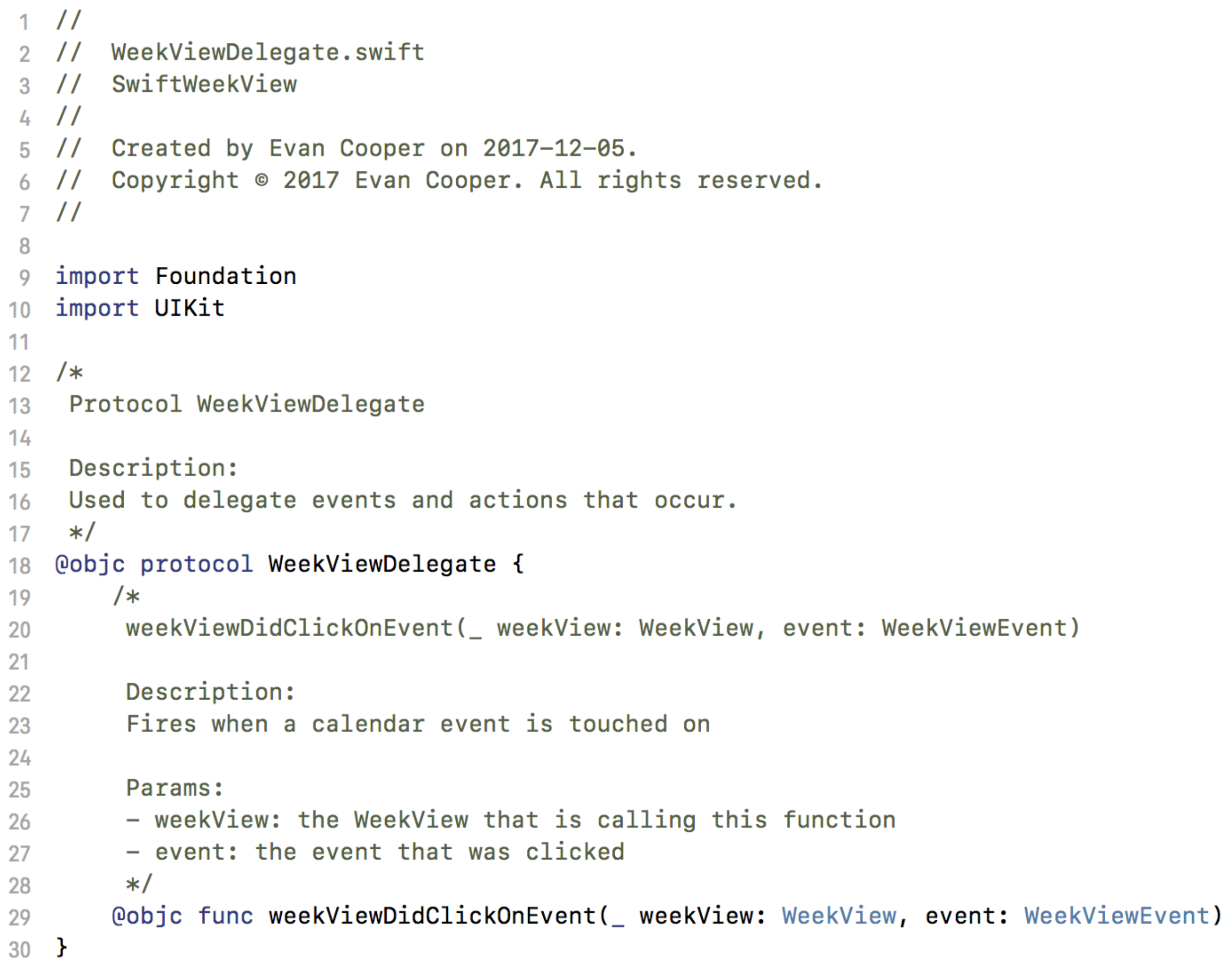


FIGURE 5 - Displaying calendar events that occur at the same time.

## INTERACTING WITH CALENDAR EVENTS

In today’s digital world, not only is it important to display data, but to interact and modify it as well. The WeekViewDelegate protocol allows for more complex interaction aside from scrolling through days of the week. Allowing users to interface with calendar events is the primary function of the protocol’s weekViewDidClickOnEvent method, and allows developers to define the nature and behaviour of the interaction. By allowing its self to handle user interaction, WeekView can receive a touch-gesture event, determine the location and impact of the touch and handle it accordingly.

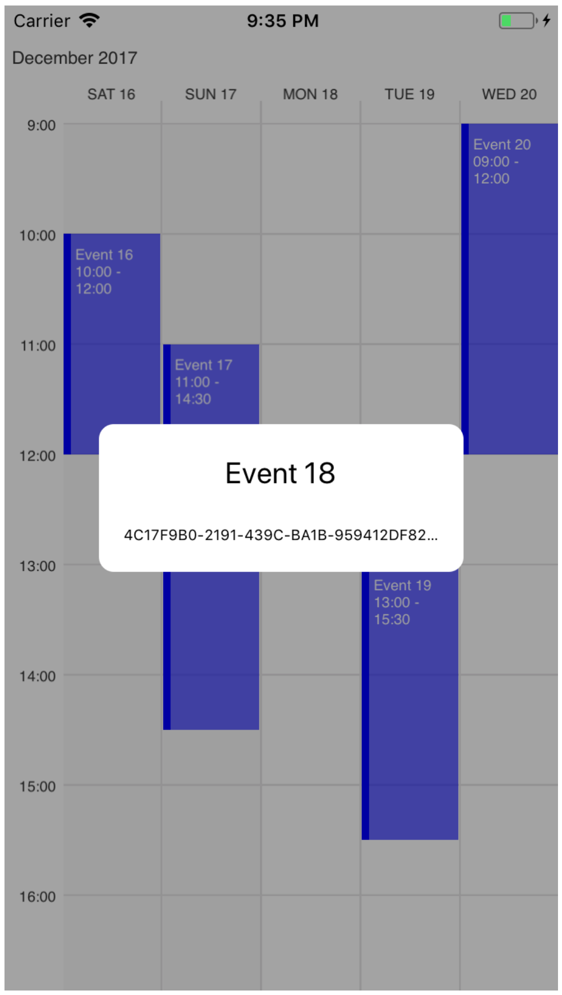


```swift
//
//  WeekViewDelegate.swift
//  SwiftWeekView
//
//  Created by Evan Cooper on 2017-12-05.
//  Copyright © 2017 Evan Cooper. All rights reserved.
//

import Foundation
import UIKit

/*
 Protocol WeekViewDelegate

 Description:
 Used to delegate events and actions that occur.
 */
@objc protocol WeekViewDelegate {
    /*
     weekViewDidClickOnEvent(_ weekView: WeekView, event: WeekViewEvent)

     Description:
     Fires when a calendar event is touched on

     Params:
     - weekView: the WeekView that is calling this function
     - event: the event that was clicked
     */
    @objc func weekViewDidClickOnEvent(_ weekView: WeekView, event: WeekViewEvent)
}
```

FIGURE 6 - WeekViewDelegate protocol definition.

After the WeekView calls on its delegate to receive each event, it will call on its styler property to create the view for the event. When events are initialized, they automatically initialize a constant unique identifier (UID) property that is used to easily differentiate its self from other events. Event views are subclasses of UIView, which are native Swift objects that manage content for a specified rectangular area on a device's screen. The subclass of UIView, WeekViewEventView, only adds a single property, which is its event's UID. Similar to an event's UID, it is fundamental to identifying the view's event from other events in the calendar. With the view now created, either through the default or custom implementation of the WeekViewStyler, a gesture recognizer is added to the view, so that it can handle interaction. In Swift, gesture recognizers "[decouple] the logic for recognizing a sequence of touches (or other input) and acting on that recognition" (Apple, 2017). Each event view's gesture recognizer will call on an internal function to WeekView, that handles the interaction and sends the information to the delegate through the weekViewDidClickOnEvent method.

Implementing the delegate method is straight forward, and requires minimal overhead. A class that conforms the WeekViewDelegate protocol will simply need to set the delegate property of the app's WeekView instance to its self, and implement the protocol method. It will be now ready to handle gestures acted on any calendar event, since the event is passed through the delegate. An implementation of the protocol can include anything from printing the event's details to the console, or presenting a new view to the user that can focus on more detailed information related to the event [Fig. 7].



Event 18

4C17F9B0-2191-439C-BA1B-959412DF82...

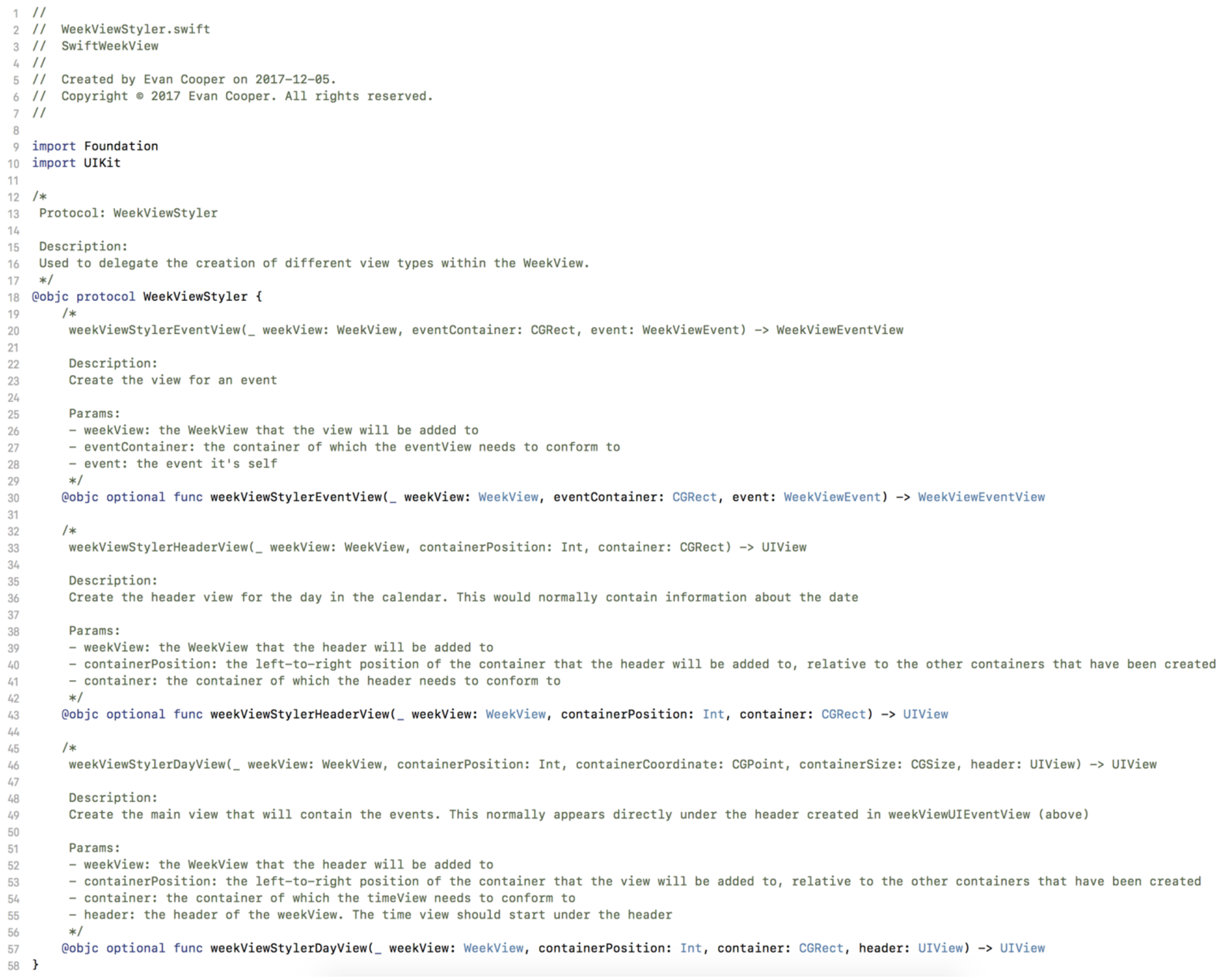
FIGURE 7 – Interacting with calendar events

## STYLING

Creating a framework with the objective for it to be used by many different developers and projects, requires the interface to be adaptable to the style of the various applications that would possibly use it. It is important to provide this ability to users of this library, as every application has unique style, and generic libraries that discourage customization will most likely be rejected during the planning phase of an application. In order to achieve this type of functionality, a WeekView implements a protocol that handles various interface components.

The WeekViewStyler protocol [Fig. 8] contains a few different and optional methods that give an implementation the ability to change any user interface component. It allows for definitions of the view that a calendar event is placed in, the column that an event is placed which represents an individual day and the column's header that by convention, displays the date. In each protocol method, there is a recurring parameter of a container that is extremely important to the usage of custom styling. When allowing users to modify the view components of the WeekView, it became apparent that some constraints to the level of customizability needed to be added. Each container provided to the protocol methods is the container that will host the new view that is being created. This means that views that are returned to any WeekViewStyler protocol methods need to have the same dimensions as the container that was provided during their function call. Otherwise, the WeekView will use its own implementation of the styling method to define the required view. Adding this verification step prevents an implementation of WeekView to be

By default, in Swift, all protocol methods are mandatory. This means that a class which conforms to a protocol must implement all the methods that are defined in the protocol. In most cases, this type of conformity is expected, however, in the context of this project, a developer who is adding the WeekView to their application may only want to modify specific parts of the view, and keep the default implementation for others. By consequence, the default implementation needs to be defined in the WeekView class to cover the case where not all protocol methods are defined by a given instance of WeekView. During initialization of each customizable view, the WeekView will check to see if it's "styler" property responds to the protocol method responsible for the creation of the given view, and call upon it to initialize the view. In the case where the styler does not respond the method, WeekView will use its own implementation of the method and provide the default functionality for view creation. This is made possible by Swift's responds(to:) method, which tests whether objects respond to specific function calls, and ultimately allows developers who use this library to modify the view freely by implementing only the protocol methods they require.



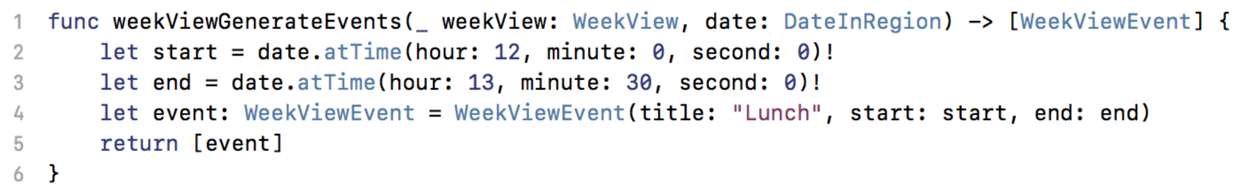
```swift
//
//  WeekViewStyler.swift
//  SwiftWeekView
//
//  Created by Evan Cooper on 2017-12-05.
//  Copyright © 2017 Evan Cooper. All rights reserved.
//

import Foundation
import UIKit

/*
 Protocol: WeekViewStyler

 Description:
 Used to delegate the creation of different view types within the WeekView.
 */
@objc protocol WeekViewStyler {
    /*
     weekViewStylerEventView(_ weekView: WeekView, eventContainer: CGRect, event: WeekViewEvent) -> WeekViewEventView

     Description:
     Create the view for an event

     Params:
     - weekView: the WeekView that the view will be added to
     - eventContainer: the container of which the eventView needs to conform to
     - event: the event it's self
     */
    @objc optional func weekViewStylerEventView(_ weekView: WeekView, eventContainer: CGRect, event: WeekViewEvent) -> WeekViewEventView

    /*
     weekViewStylerHeaderView(_ weekView: WeekView, containerPosition: Int, container: CGRect) -> UIView

     Description:
     Create the header view for the day in the calendar. This would normally contain information about the date

     Params:
     - weekView: the WeekView that the header will be added to
     - containerPosition: the left-to-right position of the container that the header will be added to, relative to the other containers that have been created
     - container: the container of which the header needs to conform to
     */
    @objc optional func weekViewStylerHeaderView(_ weekView: WeekView, containerPosition: Int, container: CGRect) -> UIView

    /*
     weekViewStylerDayView(_ weekView: WeekView, containerPosition: Int, containerCoordinate: CGPoint, containerSize: CGSize, header: UIView) -> UIView

     Description:
     Create the main view that will contain the events. This normally appears directly under the header created in weekViewUIEventView (above)

     Params:
     - weekView: the WeekView that the header will be added to
     - containerPosition: the left-to-right position of the container that the view will be added to, relative to the other containers that have been created
     - container: the container of which the timeView needs to conform to
     - header: the header of the weekView. The time view should start under the header
     */
    @objc optional func weekViewStylerDayView(_ weekView: WeekView, containerPosition: Int, container: CGRect, header: UIView) -> UIView
}
```

FIGURE 8 - WeekViewStyler protocol definition.

## DEPENDENCY - SWIFTDATE

During the course of development, it became difficult to create and manage sophisticated date objects with Swift’s native Date structure. Creating Date objects for specific times required odd calculations such as the difference in time between initialization of such an object, and the actual date that was attempting to be created. Additionally, modifying dates in Swift was even more difficult to accomplish, since it required initialization of helper objects that were difficult to understand. After attempting to work with Swift’s Date library, I decided to use a popular third-party date management library called SwiftDate. It made initialization and modification of dates extremely easy, since both could be accomplished in a single line of code. Additionally, SwiftDate is able to manage date objects across different time-zones by default, providing an added benefit that would have required valuable and significant time and effort to achieve, if Swift’s natural Date library was used.

The documentation provided with SwiftDate was also a contributing factor to it being chosen to as the date management system supporting this project. It outlines in detail how to create and modify date objects in an easy-to-read fashion. Each parameter and method are clearly and precisely explained, allowing any user of this library to quickly understand how to use this library. The documentation also explains how Additionally, the SwiftDate documentation even provides a conceptual overview that describes the entire platform, in order to provide a general understanding of its abilities.



```swift
func weekViewGenerateEvents(_ weekView: WeekView, date: DateInRegion) -> [WeekViewEvent] {
    let start = date.atTime(hour: 12, minute: 0, second: 0)!
    let end = date.atTime(hour: 13, minute: 30, second: 0)!
    let event: WeekViewEvent = WeekViewEvent(title: "Lunch", start: start, end: end)
    return [event]
}
```

FIGURE 9 - Using SwiftDate with WeekViewDataSource

Ultimately, due to SwiftDate’s ease of use, documentation and popularity, it was an easy decision for it to be included in this project as a replacement. Most of SwiftDate’s use within this project is internal, and not exposed to users. However, within the WeekViewDataSource protocol method, calendar events are expected to be returned, with each calendar event expecting a start and end date. The implementation of this requirement is straight forward and can be accomplished in minimal lines of code [Fig. 9]. I have confidence that developers who are using this library in their applications will be able to quickly and easily understand how to use SwiftDate.

## RESULTS

### LIMITATIONS

Although much has been accomplished with this project, there are some limitations to the features that the library provides.

The skeleton of the display is generated by the UIInfiniteScrollView class, who's main purpose is to allow for similar content to be scrollable through an infinite timeline, on either a horizontal or vertical plane. While this functionality is crucial to the fundamentals of the calendar's scrolling ability, the implementation has the unfortunate limitation of a single-dimensional scroll plane. In the context of a calendar, this means that the entire length of a single day must be visible at all times on the user's screen. It may be difficult for devices that have limited screen real-estate to legibly display a full twenty-four hours of possible content all at once. Fixing this issue would require UIInfiniteScrollView to allow for a limited amount of scrolling, opposite the infinite plane. Unfortunately, the time limitations of this project prohibited the ability to add such functionality.

Responding to interaction with calendar events was one of the last features that was added to the calendar view. As explained previously, since certain properties of the UIView class such as the collection of gesture recognizers would be lost when a copy of the view was created, the UIInfiniteScrollView class could not be completely independent of the WeekView class. More specifically, the UIInfiniteScrollView would have to re-add the gesture recognizers that were lost. This required the explicit re-initialization of these attributes and by consequence, forced the scroll view to choose the type of recognizer that was being added to the views beneath it. Although interaction is still possible with the calendar events, the design of the components only allows for the library to respond to one specific gesture; a tap. In order to solve this limitation, the scroll view mechanism would need to avoid copying views and should use the original instances provided by the data source protocol.

Although this library accomplishes its primary goal of being able to display calendar events in a week view, a useful feature would be the ability to change the scope of the view on demand.

More specifically, it would be useful if one could change the number of days that are concurrently visible at runtime. Currently, an instance of WeekView will be able to only show a fixed number of days at a time, and making a change to that would require re-initialization of the entire WeekView. There are many consequences, however the most critical issue would be the fact that all previously created calendar events would need to be initialized again, making this issue more severe when events are loaded from the internet. Along the same lines, fully-functional calendar applications have more than just a week view like the one provided in this library. For example, the default iOS calendar application contains a month view in which users gain an overview of the entire month. Other calendar applications, such as Microsoft's Outlook, even include overviews for an entire year. Ultimately, this library would be significantly more useful and powerful to users if they had the ability to scale the scope of their calendar from their choosing whilst keeping their data intact.

## EXAMPLE APPLICATION

In the git repository that hosts the source code for this project, there is an included example iOS application that demonstrates the abilities of this library. Alongside the documentation provided in a "readme" file and within the comments of the code, developers can reference the provided application to gain a better understanding of the implementation and usefulness of this project.

The goal of the application was not to create a feature-heavy and complete app, but to only have the necessary components required to showcase this library. The application has a single view controller with only two properties: an instance of WeekView, and a small user-interface class used for interaction with events.

The WeekView instance is initialized with as many default values as possible, and defines the respondsToInteraction value to be true, allowing for the view controller to handle interactions with the calendar events. Additionally, the view controller implements all three WeekView protocols; WeekViewDataSource, WeekViewDelegate and WeekViewStyler. It populates each day with a single event through the data source protocol. It handles event interaction by displaying a dismissible popup over the screen with information about the event with the EventDetailLauncher class, which is a is a simple class that encapsulates the ability to display and dismiss a minimalistic popup view over the entire screen. Finally, it implements custom styling by modifying only the event view with a custom view.

## CONCLUSION

In all, this project has accomplished its primary goals. Swift Week View is an original iOS library that has alleviated the missing functionality of Apple's default calendar application that is shipped with iOS devices, by being able to display calendar events in a week-view. It's default implementation visually resembles other iOS and macOS applications. By providing public protocols as high-level interfaces to the complex functionality, alongside programmatic and storyboard initialization options, this interface component has an experience of an authentic Swift interface library. After completing this project, developing the original course scheduling application that motivated the creation of the WeekView is now significantly more possible in iOS. I plan to continue the development of this project to add additional features and eliminate the limitations described earlier.

## BIBLIOGRAPHY

Apple. (2017, September 19). *Code Diagnostics Documentation*. Retrieved September 25, 2017, from Main Thread Checker: https://developer.apple.com/documentation/code_diagnostics/main_thread_checker

Apple. (2017, September 19). *Swift Documentation*. Retrieved September 25, 2017, from Protocols: https://developer.apple.com/libary/content/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html

Apple. (2017, September 24). *Swift Evolution*. Retrieved September 25, 2017, from Apple GitHub: https://apple.github.io/swift-evolution

Apple. (2017, September 19). *UICollectionView*. Retrieved September 25, 2017, from UIKit Documentation: https://developer.apple.com/documentation/uikit/uicollectionview

Apple. (2017, September 19). *UIGestureRecognizer*. Retrieved September 25, 2017, from UIKit Documentation: https://developer.apple.com/documentation/uikit/uigesturerecognizer

Apple. (2017, September 19). *UIScrollView*. Retrieved September 25, 2017, from UIKit Documentation: https://developer.apple.com/documentation/uikit/uiscrollview

CocoaPods Dev Team. (2017, September 16). *CocoaPods.org*. Retrieved September 15, 2017, from CocoaPods.org: https://cocoapods.org/

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994, November). *Design Patterns Book*. Retrieved from Gang of Four: http://wiki.c2.com/?DesignPatternsBook

malcommac. (2017, November 26). *SwiftDate Main Concepts*. Retrieved from SwiftDate: https://malcomman.github.io/SwiftDate/main_concepts.html