Niko Holopainen

Reactive iOS Development with RxSwift

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Bachelor’s Thesis

09 February 2022

Abstract

Author:	Niko Holopainen
Title:	Reactive iOS Development with RxSwift
Number of Pages:	37 pages
Date:	09 February 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Mobile Solutions
Supervisor:	Patrick Ausderau, Senior Lecturer

Over the last decade, demand for reactive and scalable mobile applications has massively expanded. Reading developer news about scalable applications certainly brings up the benefits of Reactive Extensions (Rx). RxSwift library, which is part of the larger Rx family, allows using Swift in an entirely new way. Writing rather difficult asynchronous code in Swift becomes more straightforward using RxSwift.

This study aims to explain the architectural concepts of reactive programming in iOS development. This study also proposes the usage of RxSwift combined with Model-View-ViewModel (MVVM) design pattern to create robust iOS applications. In addition, the study offers an alternative way of building user interfaces without iOS Storyboards.

During the study, an iOS application was developed with Swift emphasizing the usage of RxSwift with MVVM design pattern. The resulting application fetches cryptocurrency data from CoinGecko Application Programming Interface (API) and presents the data using graphs. A real-time WebSocket chat was also created to showcase reactive methodologies. User interface for the application was built with UIKit using SnapKit to provide autolayout capabilities. Created exclusively for the study, the application is not released to the App Store nor is the source code publicly available.

The study revealed that while RxSwift is an excellent choice for skilled developers, high learning curve and the needed knowledge of the Swift programming language can be challenging. Also, the possibility of introducing strong reference cycles and misbehaving streams can be challenging to debug.

Keywords: RxSwift, Reactive programming, iOS, MVVM, mobile

Tiivistelmä

Tekijä:	Niko Holopainen
Otsikko:	Reaktiivinen iOS-kehitys RxSwiftillä
Sivumäärä:	37 sivua
Aika:	9.2.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mobile Solutions
Ohjaaja:	Lehtori Patrick Ausderau

Viime vuosikymmenen aikana reaktiivisten ja skaalautuvien mobiilisovellusten kysyntä on kasvanut valtavasti. RxSwift-kirjasto, joka on osa suurempaa Reactive Extensions (Rx) -kirjastoa, mahdollistaa Swift-ohjelmointikielen käytön aivan uudella tavalla. RxSwift mahdollistaa yksinkertaisemman asynkronisen koodin luonnin Swift-ohjelmointikielellä.

Insinöörityön tarkoituksena oli perehtyä reaktiivisen ohjelmoinnin arkkitehtonisiin käsitteisiin iOS-kehityksessä. Työssä tutkittiin RxSwiftin käyttöä yhdessä Model-View-ViewModel (MVVM) -suunnittelukuvion kanssa iOS-sovellusten luomiseksi. Työssä tutkittiin myös vaihtoehtoista tapaa rakentaa käyttöliittymiä ilman iOS-käyttöjärjestelmän tavanomaisia menetelmiä.

Työn aikana kehitettiin iOS-sovellus, joka perustui RxSwiftin ja MVVM-suunnittelukuvion käyttöön. Tuloksena syntynyt sovellus noutaa kryptovaluuttatietoa avoimesta rajapinnasta ja esittää tiedon käyttämällä kaavioita. Lisäksi luotiin reaaliaikainen keskusteluominaisuus esittelemään reaktiivisia menetelmiä. Sovelluksen käyttöliittymä rakennettiin UIKit-kehikolla, johon yhdistettiin kolmannen osapuolen SnapKit-kirjasto, joka tarjoaa yhtenäisen tavan luoda automaattisen asettelun ominaisuuksia. Yksinomaan insinöörityötä varten luotua sovellusta ei julkaistu sovelluskauppaan eikä lähdekoodi ole julkisesti saatavilla.

Insinöörityö osoitti, että vaikka RxSwift on erinomainen valinta taitaville iOS-kehittäjille, kirjaston korkea oppimiskäyrä ja tarvittava Swift-ohjelmointikielen osaaminen voivat olla haasteellisia uusille iOS-kehittäjille. Myös muistivuotojen ja epätoiminnallisten datasekvenssien virheenkorjaus saattaa olla haastavaa.

Avainsanat: RxSwift, reaktiivinen ohjelmointi, iOS, MVVM, mobiili

Contents

List of Abbreviations

API:	Application Programming Interface
ARC:	Automatic Reference Counting
DSL:	Domain Specific Language
HTTP:	Hypertext Transfer Protocol
IDE:	Integrated Development Environment
MVC:	Model-View-Controller
MVVM:	Model-View-ViewModel
Rx:	Reactive Extensions
SDK:	Software Development Kit
TCP:	Transmission Control Protocol

1 Introduction

To create reactive and robust mobile applications, the need to handle multitude of concurrent tasks grows significantly. The tasks can vary from playing audio, using the camera, making network calls to handling user interface input. Passing data from one process to another or even just observing that tasks happen in the correct sequence asynchronously, can cause the developer a lot of trouble.

Before the announcement of Combine in 2019 for iOS and Kotlin Flow in 2020 for Android, developers had to rely on the relatively complicated native methodologies to support asynchronous task scheduling [1, p. 16]. Even after the introduction of these frameworks, the problem of learning and maintaining knowledge of multiple platforms and syntax for cross-platform development persists. Furthermore, supporting older iOS devices with Combine, such as the iPhone 6, is impossible due to the minimum target level being available from iOS 13 onwards [2].

For many mobile developers, Reactive Extensions (Rx) library offers an alternative way of building modern mobile applications. Similar reactive syntax correlates into handling both native mobile platforms almost uniformly. For a mobile developer, the capability of handling iOS and Android development is very valuable. Therefore, learning Rx enables developers to quickly learn new programming patterns and methodologies.

RxSwift combines practises from imperative and declarative programming to create solutions for side effects and state handling. It enables the developer to write more efficient code and utilize the full potential of reactive programming by improving architectural concepts such as code isolation, reusability, and decoupling. [1, p. 28]

2 RxSwift

Back in 2009, a team at Microsoft solved the challenge of asynchronous and reactive application development thus creating the first ever Rx-library, Rx.NET. By going open source in 2012, it permitted other platforms to reimplement the same functionality, therefore making Rx into a cross-platform standard. Nowadays, almost all mainstream programming languages have the support of Rx. [1, p. 31-32]

Since Rx is an extension of the observer pattern, it supports data sequencing, event handling and adds operators that enable declarative programming style. Furthermore, it tackles the problem of mutable state variables by reacting to events.

RxSwift is a pure Swift library for iOS, macOS, tvOS and watchOS. It allows for parameterized runtime execution via schedulers for asynchronous, event-driven code by using observable streams and functional style operators. In its essence, RxSwift simplifies developing asynchronous programs by reacting to new data and process it in a sequential and isolated manner. [3]

2.1 Asynchronous code

Almost all native Swift code, whether it be UI events or network requests, executes some work asynchronously. An iOS application, at any moment, might be reacting to tap gestures, playing audio, uploading a large image, or saving data to local storage. None of these tasks block each other’s execution. iOS provides multiple different Application Programming Interfaces (API) that allow for asynchronous and multithreaded execution. However, maintaining code that is truly running concurrently with native methodologies is complicated. [4, p. 27]

Apple provides multiple APIs in the iOS Software Development Kit (SDK) that help writing asynchronous code. Some of the most used APIs are `NotificationCenter` and `UIApplicationDelegate`.

`NotificationCenter` runs a code block whenever an event of interest occurs, such as taking a screenshot or performing a 3D Touch. `UIApplicationDelegate` defines an object which acts on behalf, or in coordination with, another object, such as receiving a remote notification in `AppDelegate`. [1, p. 24-26]

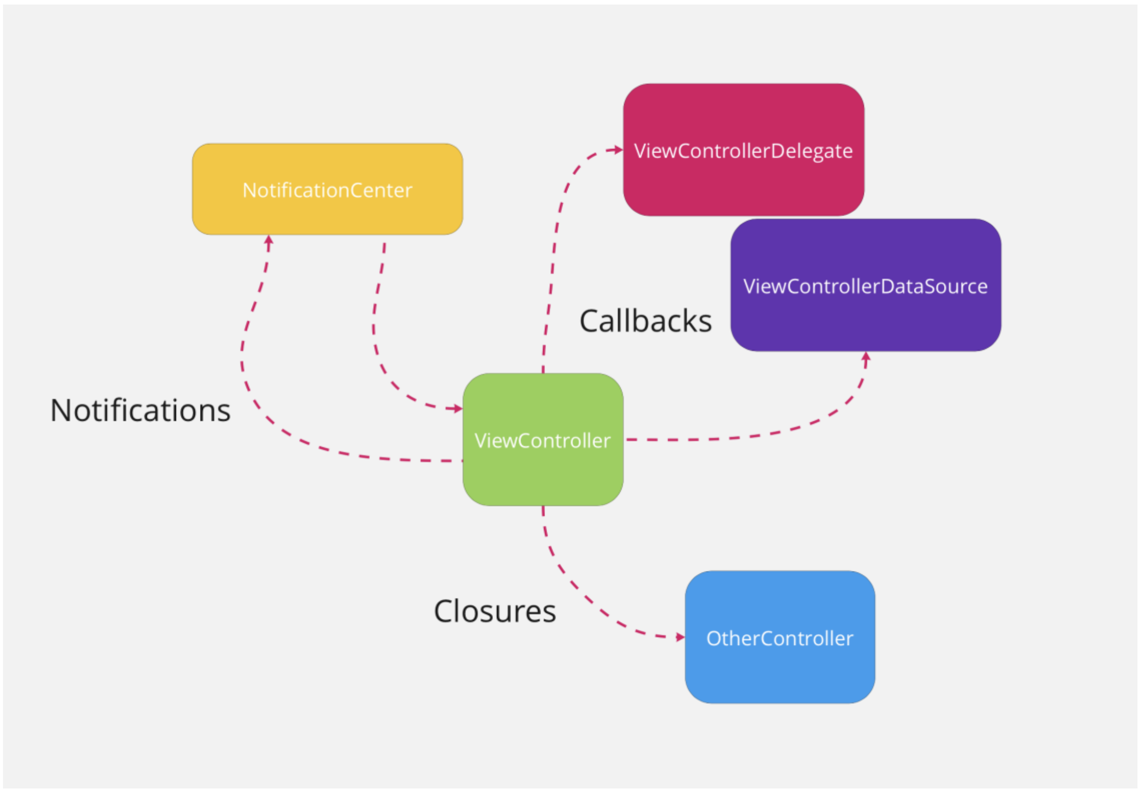


Figure 1. Example of API usage within a UIViewController. [1, p. 26]

The hierarchy of implementing delegates, passing closures, and subscribing to notification events is presented in Figure 1. To handle all three cases, each one requires its own pattern to be implemented. Since none of the APIs share universal language, the need to implement multiple delegates and patterns becomes complicated.

2.2 Observable

Observables are the primary building blocks of RxSwift. Provided from the `Observable<T>` class, where the T represents a generic data type, produces sequences of asynchronous events that hold immutable values. Sequences can be visualized easily by a stream of values within a period. [4, p. 35]

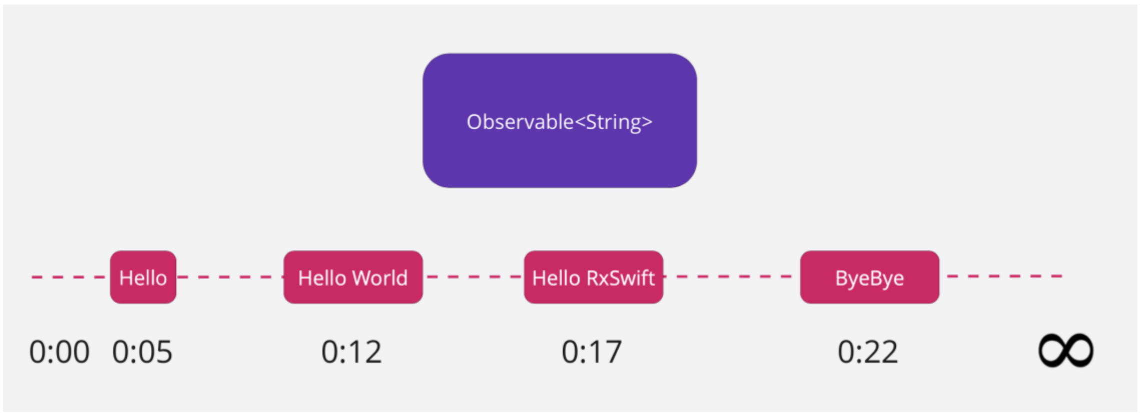


Figure 2. Stream of String values within a period. [4, p. 35]

The stream of observables can hold multiple values. In Figure 2, the value emitted at each timestamp is different. Duration of the stream can be assumed to be infinite. Using event sequences within the observable contract, decoupling code becomes automatically easier. Since the contract does not manage either the observable or observer, the need to implement any delegates or closure injecting becomes irrelevant.

2.2.1 Subscribing

Observable allows objects and/or consumers to subscribe for events or values emitted by another object over time and process those events in real time. The `ObservableType` protocol, which `Observable<T>` implements, can emit four types of events, `onNext`, `onCompleted`, `onError` and `onDisposed`. The next event emits the latest value provided from the observable. This way the observer collects values from the sequence and continues to do so indefinitely, until a terminating event is emitted. The completed event terminates the emitting of values in a successful manner. Completion also indicates that the observable has completed its lifecycle. The error event terminates the emitting of values and pushes a corresponding error to the sequence. This also indicates the end of the lifecycle. Lastly, the disposed event which indicates that the subscription has been deallocated properly. The disposed closure will always execute as the last event of a sequence. [5]

```
ApiService.instance.getCryptoCurrencies()
    .subscribe(onNext: { cryptos in
        // [Crypto]
    }, onError: { error in
        // Error
    }, onCompleted: {
        // Completed
    }, onDisposed: {
        // Disposed
    }).disposed(by: disposeBag)
```

Listing 1. Subscribing to an observable and handling all four event types.

An asynchronous function, which performs a network request and returns a collection of cryptocurrencies is presented in Listing 1. The result of the function is then subscribed to and handled as events are emitted. Once the network request is finished in a successful manner, the `onNext` closure emits an object. In case of an error, the `onError` closure executes and the subscription will terminate. Lastly, the `onCompleted` closure will terminate the subscription once the function no longer emits new values and the `onDisposed` closure indicates that the subscription is disposed properly. An observable will never emit events or execute closures before its subscribed to. [6]

Sending a network request, which is mapped to a function that returns an observable, returns a single sequence of data, and terminates afterwards. Subscribing to user interface events, will not terminate, unless explicitly told to end or deallocated from the memory.

As an observable is subscribed to, the number of values emitted can vary from zero to infinite. Network requests, which are expected to terminate either successfully or unsuccessfully, are considered as finite sequences. Additionally, user-initiated events such as device orientation or button taps, are considered as infinite sequences. [7]

```swift
switcher.rx.isOn
    .subscribe(onNext: { value in
        switch value {
        case true:
            print("Switch is ON")
        case false:
            print("Switch is OFF")
        }
    }).disposed(by: disposeBag)
```

Listing 2. Subscribing to a sequence of UISwitch events.

Subscription to an infinite sequence of `UISwitch` events is presented in Listing 2. The sequence of the emitted values does not have an ordinary end. During its lifecycle, the switch might not be toggled at all, thus not emitting any events. However, the sequence is not terminated because the switch is allocated, and the subscription is existing. As toggling the switch can only emit `onNext` or `onCompleted` events, the `onError` closure is not available in the context. The switch has an initial value of false.

2.2.2 Disposing

The Swift programming language uses Automatic Reference Counting (ARC) for memory management. When creating a class instance, ARC allocates memory that is used to store object data. As with all programming languages, system resources are limited, and any used memory must be freed after use. [8]

Importance of memory management is further emphasized when using RxSwift. When any subscription is to be freed from memory, it must be deallocated properly. Managing each subscription independently would become complicated.

All subscriptions are of type `Disposable`, which represents a disposable resource. A `DisposeBag` object holds references to `Disposable` objects and properly deallocates them when a subscription has received its last value, the subscription errors or completes, or when the `DisposeBag` object gets deallocated. `DisposeBag` usually exists within a `UIViewController` and deallocates with the `UIViewController` object. Each subscription should be disposed either manually or automatically. This way, each subscription deallocates properly, unless a memory leak is present. [9]

```swift
deinit {
    print("Deallocating UIViewController")
}

override func viewDidLoad() {
    super.viewDidLoad()

    ApiService.instance.getCryptoCurrencies()
        .subscribe(onNext: { cryptos in
            // [Crypto]
        }, onError: { error in
            // Error
        }, onCompleted: {
            // Completed
        }, onDisposed: {
            // Disposed
        }).disposed(by: disposeBag)
}
```

Listing 3. Disposing a subscription with the DisposeBag object.

The full lifecycle and usage of `Disposable` is presented in Listing 3. Without properly disposing a disposable, the Swift compiler warns of the unused subscription. RxSwift will automatically handle all disposable resources, if the `DisposeBag` object is used properly.

2.3 Subjects

Observables cannot emit values which are injected during runtime. To manually push new events to a stream, a subject needs to be used. Subject can behave as an observable or observer. Each time a subject receives an event, it broadcasts the emitted value to its subscribers. Some subjects can be initialized with a buffer size or initial value [10]. Different types and traits of subjects and relays are presented in Table 1.

Table 1. Trait comparison between subjects and relays. [10]

Type	Initial value	Buffer size	Emits only new values	Can error	Library
PublishSubject	No	No	Yes	Yes	RxSwift
ReplaySubject	Yes	Yes	No	Yes	RxSwift
BehaviorSubject	Yes	No	No	Yes	RxSwift
PublishRelay	No	No	Yes	No	RxCocoa
ReplayRelay	No	No	Yes	No	RxCocoa
BehaviorRelay	Yes	No	No	No	RxCocoa

The RxCocoa library offers a `Relay` type, which is a wrapper for subject. RxCocoa is a reactive library which depends on RxSwift. It enables the usage of Cocoa APIs found within the iOS SDK in a reactive manner. Cocoa includes frameworks such as the Foundation, which is mandatory and the only environment to develop iOS applications with. Relays share similar traits as subjects but cannot error or complete so the `onError` and `onCompleted` closures are not available. As with subjects, some relays cannot emit events before subscribed to. [11]

```
1  let relay = PublishRelay<String>()
2
3  relay.accept("This wont emit")
4
5  relay
6      .subscribe(onNext: { value in
7          print("Got value: \(value)")
8      }).disposed(by: disposeBag)
9
10 relay.accept("This will emit")
```

Listing 4. Subscribing to a PublishRelay and emitting events.

The allocation, subscribing and value injecting of a `PublishRelay` is presented in Listing 4. Due to `PublishRelay` requiring a subscription before it starts emitting values, the first accept trigger on line 3 will not emit any events. Afterwards, the relay is subscribed to and accept triggered once again, resulting in a value in the `onNext` closure.

2.4 Operators

Operators are the key elements of RxSwift. The ability to filter, create and manipulate data sequences enable reactive paradigms. Like traditional mathematic functions such as addition and subtraction, operators can be used in a similar way. Operators can be split into three main categories, transforming operators, filtering operators, and combining operators. Beyond these operators, some subcategories exist. Operators such as, utility, mathematical and conditional are excluded from the study, due to their minimal usages. [12]

Connecting all three operator types to observable sequences, gives RxSwift its declarative and reactive benefits. Within any observable sequence, an operator can transform, filter, or combine the current value. Any operated sequence is dependant of its previous operation. Generally, standard filter operators do not mutate the original sequence. Transforming operators are the most used feature in RxSwift. Swift provides similarly named functions such as `map()`, `flatMap()` and `compactMap()` within collection variables. However, these functions are not available in observable closures nor behave similarly. [13]

```swift
let id = PublishSubject<String>()
let button = UIButton()

    Observable.combineLatest(
        button.rx.tap.debounce(.milliseconds(300),
        scheduler: MainScheduler.instance),
    id) { ($1) }
    .filter { !$0.isEmpty }
    .flatMapLatest {
        ApiService.instance.getCryptoCurrencies(ids: $0)
    }
    .subscribe(onNext: { cryptos in
        // [Crypto]
    }, onError: { error in
        // Error
    }).disposed(by: disposeBag)
```

Listing 5. Transforming a button tap and a String into a collection of cryptocurrencies.

Utilization of operators combined with subscribing is presented in Listing 5. Value and type of the sequence is transformed and filtered during its lifecycle. Each time a button tap is registered, or the subject value changes, the sequence is triggered. In case of the filter operators predicate fails, the sequence comes to a hold and stops emitting any events. However, the sequence contains an active subscription, thus not disposing.

To prevent multiple network requests, tap gesture of the button is throttled to only emit events outside a time interval. Any overlapping tap gestures are ignored and only one is processed onwards in the sequence. Likewise, the network request itself is mapped inside an operator, which ensures that any ongoing requests are cancelled prior to performing.

Lastly, in the subscribe closure, either a value is emitted, or an error has occurred. By chaining operators in sequences, all values are expected to represent their latest states. Sequences can be transformed multiple times and are independent of each other. The original button tap, and subject value can be transformed into something else with another subscription, while keeping the emitted cryptocurrency collection completely decoupled.

Creating extensions for `ObservableType` or `UIView` components are treated equally as the traditional Swift extension syntax. Custom reactive extensions can conform to multiple protocols and generic types. Furthermore, extensions can be used in more concise and effective way to produce more maintainable code.

```
extension Reactive where Base: UIView {
    var tapGesture: Observable<Void> {
        let tapGesture = UITapGestureRecognizer()
        base.isUserInteractionEnabled = true
        base.addGestureRecognizer(tapGesture)
        return tapGesture.rx.event.mapToVoid()
    }
}

extension ObservableType {
    func mapToVoid() -> Observable<Void> {
        return self.map { _ in () }
    }

    func mapTo<C>(constant: C) -> Observable<C> {
        return self.map { _ in
            return constant
        }
    }
}
```

Listing 6. Custom extensions for UIView and ObservableType.

Extensions for observable sequences and the `UIView` class are presented in Listing 6. The `mapToVoid()` function removes boilerplate code by encapsulating the `map()` function, which returns an empty tuple. The `Observable<C>` represents any generic data type and the helper function `mapTo()` returns any constant given as a parameter.

To extend the Rx namespace for view components, a `Base` class needs to be implemented. The extension can be used in any component, which implements the type of the `Base` class. All observable sequences implement the `ObservableType` protocol, thus being without any `Base` class implementations.

2.5 Schedulers

By default, all components within an iOS application run single threaded. All executed tasks, whether it be user interface event or saving data to user preferences, runs on the main thread. Scheduler is a decoupled context, usually a thread, in which processes take place in synchronous or asynchronous order. Common misunderstanding of schedulers is that they behave uniformly with threads. Since a scheduler is unaware of any relation between itself and the current thread, it is necessary to verify the context the scheduler is operating in. The ability to perform tasks concurrently using different threads and schedulers, independent of each other’s work, can be problematic. [14]

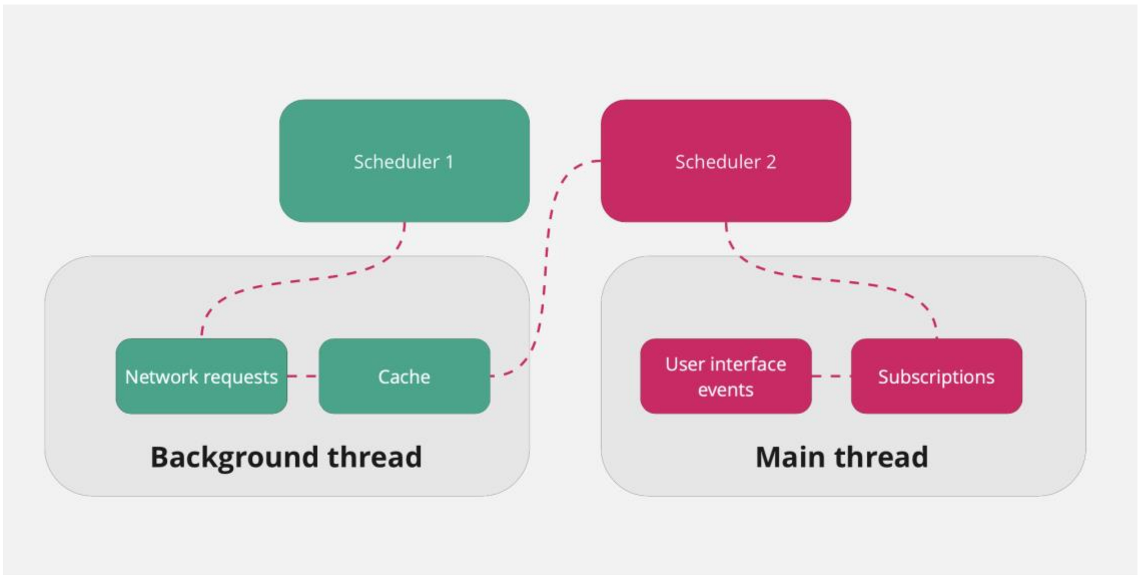


Figure 3. Visualization of two schedulers executing asynchronous workloads. [14]

The separation of background and main thread tasks into two schedulers is presented in Figure 3. Scheduler 1 performs network requests, persists the data on local cache and forwards data onwards within a background thread. Scheduler 2 receives data from the cache asynchronously, emits events to subscribers and performs user interface updates within a main thread. All user interface events must be executed from the main thread.

Within observable sequences, it is possible to switch schedulers with some limitation related to non-thread-safe objects emitted within the sequence. Non-thread-safe objects include mutable single scheduler events or values. These objects cannot be exposed across multiple threads concurrently. However, it is possible to expose such a value using RxSwift, but that would lead in to violating the core functionality of the application. [15]

```
let gScheduler = ConcurrentDispatchQueueScheduler(
    queue: .global()
)

Observable.just("bitcoin")
    .subscribe(on: gScheduler)
    .flatMapLatest {
        ApiService.instance.getCryptoCurrencies(ids: $0)
    }
    .observe(on: MainScheduler.instance)
    .subscribe()
    .disposed(by: disposeBag)
```

Listing 7. Switching schedulers during an observable sequence.

In Listing 7, the network request is processed on Scheduler 1, Figure 3, using a background thread and the subscribing observer is processed on Scheduler 2, Figure 3, using the main thread. In case of the network request performing additional transforming operators within the function call, the operators must also implement and validate the correct scheduler.

All scheduler and thread switching must be implemented and maintained by the developer. If the threading is not specified correctly, the observable sequence will execute tasks within the original thread, from which the sequence started. Not handling the possible side effects of poorly managed thread and scheduler execution is known as the hot and cold observables.

Hot observables represent a case of not having any side effects during a subscription, yet it has its own context in which events are generated. By default, the context is not accessible to RxSwift nor is it able to determine any scheduler or thread related information of itself.

In opposition to hot observables, cold observables can contain multiple side effects and does not produce any events until having a valid subscription by a subscriber. Therefore, cold observables lack any context of scheduling or threading, and will not produce events until a subscription is made.

Typically, the need to handle unexpected side effects are signs of bad architectural design. Performing any side effects upon subscription correlates to the source observable not being shared. Otherwise, all side effects are shared between subscribers. [16]

2.6 RxCocoa

RxSwift in its essence is unaware of any connection between observables and the user interface. To provide user interface control between RxSwift and UIKit, AppKit and WatchKit, a third layer is needed, RxCocoa. Each platform has its own interpretation of reactive wrappers, providing a set of built-in extensions to user interface controls. [17]

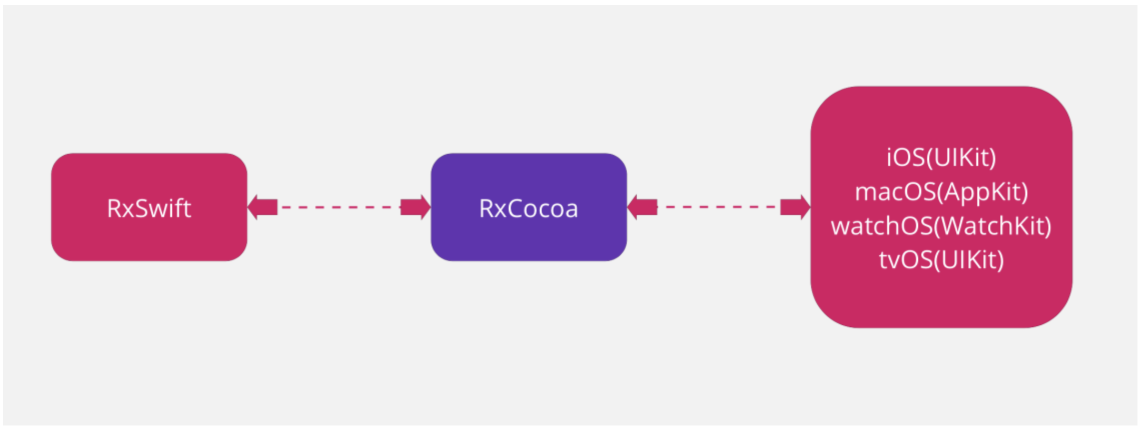


Figure 4. Linking RxSwift to platform specific user interactions using RxCocoa. [6]

The hierarchy of RxCocoa between platform specific implementation is presented in Figure 4. RxCocoa connects reactive methodologies to user interface actions between all Apple application development environments. Most importantly, RxCocoa provides reactive traits which allow user interaction events to be transformed into observable sequences and expose bindable sinks for reactive sequences. Provided traits include control events, control properties and binders.

Control events provide user interaction specific traits such as button taps or scroll view scrolling. Control properties describe values that a user interface element might have such as text or switch state. Binders work in opposite way from control events and properties, allowing to bind reactive sequences into user interface elements such as a button alpha value or enabled state. By utilizing binders, the user interface will always display its latest state.

RxCocoa provides bindable properties for almost any user interface element. For example, the reactive, non-delegate `UITableView` binding, which allows observable sequences to be bound into a table view. By combining operators found from RxSwift, validation logic for inputs, outputs, custom throttling logic and retry logic becomes straightforward. The developer can leave delegates and data sources to be handled by Rx. Building a custom table view logic using native methodologies, which includes network requests, throttling logic, and filtering logic, would result into unnecessarily complicated, hard to maintain code. [18]

```swift
let tableView = UITableView()

Observable.just("bitcoin")
    .flatMapLatest {
        ApiService.instance.getCryptoCurrencies(ids: $0)
    }.bind(to:tableView.rx.items(
                cellIdentifier: "Cell",
                cellType: CryptoCell.self)
    ) { row, item, cell in
        cell.cryptoName.text = item.parsedName
        cell.cryptoPrice.text = item.parsedPrice
    }.disposed(by: disposeBag)
```

Listing 8. Binding a collection of observable cryptocurrencies into a table view.

Implementation of binding observable sequence into a table view is presented in Listing 8. The Rx namespace is provided from RxCocoa and it enables the usage of reactive extensions for the table view. By passing in the cell identifier and the cell class, Rx will handle all dequeuing and data source implementations. An anonymous lambda function holds the row index, cell type and the cell object itself. Like all observable sequences, binding returns a `Disposable` object.

3 Architecture and protocols

3.1 MVC and MVVM

For a long time, Apple has recommended Model-View-Controller (MVC) as the architectural design pattern for iOS applications. The MVC design pattern splits and assigns components of an application to three roles, Model, View, and Controller. [19]

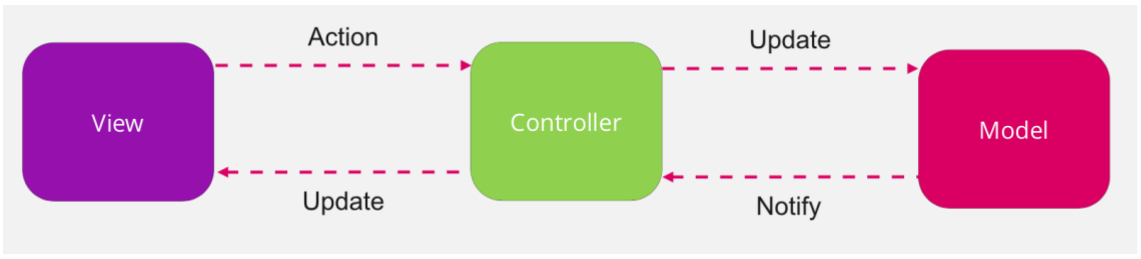


Figure 5. MVC design pattern. [19]

The hierarchy of MVC design pattern is presented in Figure 5. The Controller represents an instance of the `UIViewController` class. Controller acts as the central role since it updates both the View and Model. View is a representation of the `UIView` class, and it will only display data and forward actions such as button taps to the Controller. Lastly, the Model represents a generic service which performs network requests or reads and writes data to persist the application state.

MVC in its essence, provides simple design pattern that is somewhat functional and easy to start developing iOS applications with. In larger scale applications, classes might not be either views or models, thus being controllers. A common pitfall occurs when the controller gets overwhelmed with view and model logic creating hard to maintain code. Furthermore, MVC lacks scalability and testability regarding unit testing. Overloading any class is generally bad practice, and not necessarily a flaw in the MVC pattern. [20]

The preferred design pattern to use with RxSwift and RxCocoa is Model-View-ViewModel (MVVM). MVVM adds an additional role to the MVC pattern, ViewModel. For many iOS developers the MVVM design pattern is welcomed due to easily solving numerous issues from MVC. [1, p. 390]

By separating the view lifecycle from the business logic layer, MVVM increases the testability and scalability of the code. As the ViewModel holds all the business logic and is decoupled from the presentation layer, it can be re-used to migrate an iOS application to macOS, watchOS or even tvOS. [1, p. 390-392]

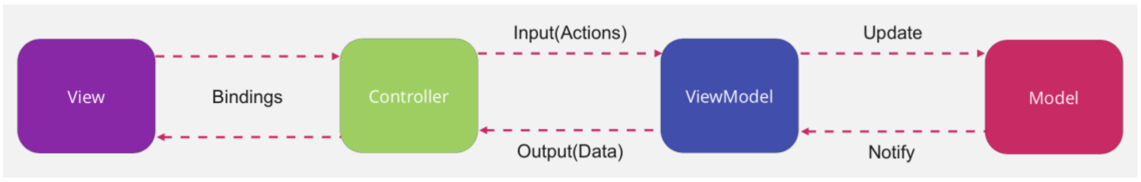


Figure 6. MVVM design pattern within an iOS application. [1, p. 391]

The hierarchy of MVVM design pattern is presented in Figure 6. Typically, in an iOS application, the MVVM term also includes a Controller. The ViewModel takes the central role as it is responsible for creating the business logic layer. It forwards data from the Model and is exposed to the Controller. Model is only exposed to the ViewModel. Controller binds the needed properties from ViewModel and View. As with MVC, View only displays the data and exposes needed actions. Like in the MVC design pattern, the Controller and View in the MVVM design pattern represent their MVC counterparts.

3.2 Libraries and frameworks

3.2.1 Pusher

Hypertext Transfer Protocol (HTTP), founded in 1991, was created to provide two-way communication between a client and a server. To this day, HTTP is used widely within web browsers and other applications that pool data from a server. However, to handle real time or push-based communication between client and a server can be challenging without wasting request resources. [21]

WebSocket is an application protocol built on top of Transmission Control Protocol (TCP). It provides the ability to communicate between a client and a server and vice versa, without performing continuous HTTP requests. To a certain degree, WebSockets behave similarly as the observer pattern, by notifying all connected observers of new events. Integrating WebSocket functionality to an application which utilizes Rx is simple since they both share similar attributes from the observer pattern. [21]

Pusher is a real time hosted API service utilizing WebSockets. PusherSwift, which is an iOS implementation of Pusher, is a real time API which provides bi-directional binding between the client and server. The effect of bi-directional API enables coupling data from two processes, in this case, the client and the server. Each end will have its own interpretation of the API and the binding happens by matching authentication keys and secrets. Authentication parameters are provided from the Pusher Dashboard. [22]

Pusher offers solutions for almost all mainstream programming languages both in client and server side. Applications that require real time communication features such as messaging, notifications, or location tracking, can be implemented with Pusher. [22]

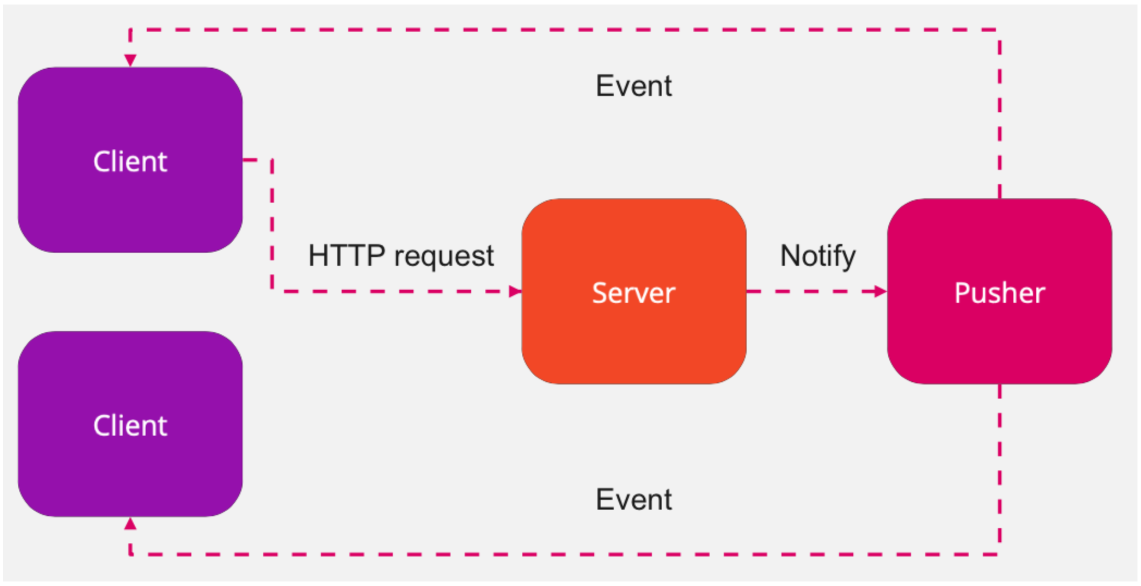


Figure 7. Implementation of bi-directional communication between Pusher, iOS client and a server. [22]

The ordinary implementation of bi-directional communication is presented in Figure 7. Whenever an action of interest, such as sending a message, occurs within the iOS client, it performs a HTTP request to the server. The server then acknowledges the event and forwards it to the Pusher API. Lastly, Pusher notifies any connected observers, in this case the iOS clients, with an event, which is then handled within the clients. The implementation remains the same regardless of the number of connected observers.

3.2.2 UIKit and SnapKit

UIKit is a framework for constructing and managing event-driven user interface within iOS and tvOS applications. It provides view hierarchy to implement multiple functionalities regarding user input and user interface events. Furthermore, UIKit enables access to native animations, device specific window information and accessibility features. [23]

When using iOS storyboards, the constraints can be visually set and inspected. However, due to numerous problems regarding merge conflicts, segue preparations, and unexpected element behavior with storyboards, building the constraint and layout hierarchy with UIKit combined with SnapKit offers a great alternative. [24]

SnapKit is a Domain Specific Language (DSL) library which enables easier use of Auto Layout and constraint capabilities. It provides better, human-readable syntax of `LayoutConstraint` class, which allows to create constraints with code. Furthermore, it removes repeating code blocks used with the `LayoutConstraint` class. Auto Layout describes hierarchy and constraints between views. [25]

```
cryptoListView.translatesAutoresizingMaskIntoConstraints = false

LayoutConstraint.activate([
    cryptoListView.leadingAnchor.constraint(
        equalTo: view.leadingAnchor),
    cryptoListView.topAnchor.constraint(
        equalTo: view.topAnchor),
    cryptoListView.trailingAnchor.constraint(
        equalTo: view.trailingAnchor),
    cryptoListView.bottomAnchor.constraint(
        equalTo: view.bottomAnchor)
])
```

Listing 9. LayoutConstraint variant of constraining a view.

The default, `LayoutConstraint` class variant of constraining a view to the edges of the screen is presented in Listing 9. The code itself is unnecessarily long and complex to read. The SnapKit variant of implementing the same logic is presented in Listing 10, which is more readable and concise.

```
cryptoListView.snp.makeConstraints { make in
    make.edges.equalToSuperview()
}
```

Listing 10. SnapKit variant of constraining a view.

SnapKit also allows for calculation of available screen estate, Auto Layout priority setting and less or greater operators to support views that might alter layout properties. The default constraint maker function is animatable, meaning it can be called within an animation closure to create visual effects during the layout constraining. Furthermore, the constraint maker function supports remaking constraints during runtime.

4 Practical implementation

The project consisted of two key parts, an iOS application as the client and a Node.js server. The client offers a convenient way of inspecting cryptocurrency prices and historical data. Furthermore, it enables a chatting option with other users. The server offers a bi-directional API provided by Pusher. For the sake of simplicity, the server does not forward any data to a database, nor does it run on any online server.

The application was written in Swift 5.5, RxSwift and RxCocoa 6.2 using Xcode 13.2 as the Integrated Development Environment (IDE). JavaScript combined with Node.js was used for the server. Dependencies were installed with Cocoapods 1.11.2.

Cocoapods is a dependency manager for all languages running on the Objective-C runtime. It resolves dependencies between installable libraries and generates a workspace which holds a collection of projects namely to include the installed dependencies. [26]

```
platform :ios, '14.5'

inhibit_all_warnings!

target 'cluster' do
   pod 'RxSwift', '6.2.0'
   pod 'RxCocoa', '6.2.0'
   pod 'RxAlamofire'
   pod 'RxKeyboard'
   pod 'PusherSwift'
   pod 'SnapKit'
   pod 'R.swift'
   pod 'Kingfisher'
   pod 'Charts'
end
```

Listing 12. Contents of a Podfile.

Cocoapods is installed via Ruby and can be integrated to any iOS, tvOS, watchOS or macOS project. To install needed dependencies, a Podfile needs to be generated and configurated. Content of the applications Podfile is presented in Listing 12. Podfile describes the project specific dependencies and installs them.

4.1 Generic API service

As the application performs multiple network requests, a maintainable and scalable API service was created. The `ApiService` class is treated as a singleton, thus only one instance of the class is allocated. The initial function returns an observable HTTP response combined with generic data provided from the response.

```swift
private func apiRequest(
    _ host: ApiHost = .coinGecko,
    _ method: HTTPMethod,
    _ path: String,
    _ headers: HTTPHeaders?,
    _ params: Parameters?,
    _ body: Data? = nil,
    _ encoding: URLEncoding = .default
) -> Observable<(HTTPURLResponse, Data)> {
    guard
        let url = URL(string: host.apiHost + path)
    else { return Observable.never() }
    return RxAlamofire.requestData(
        method,
        url,
        parameters: params,
        encoding: encoding,
        headers: headers
    )
}
```

Listing 13. Initial generic HTTP request function.

The builder request function is presented in Listing 13. It utilizes the RxAlamofire library, which provides reactive extensions to Alamofire. Alamofire is a pure Swift library used for HTTP networking requests. After the initial request responds, the incoming generic data can be decoded into a `Struct`, which is a value type declared in the application. If the request responds with empty or irrelevant data, the response tuple can be mapped to check for valid response status codes. In case of an error, the function emits a next event to an observer which displays a message, indicating a failure occurred within the request and returns with an empty tuple. Currently, the function does not support any retrying logic nor the ability to perform work after losing network connection.

```
func getCryptoCurrencies(
    ids: String = ""
) -> Observable<[Crypto]> {
    let path = "/v3/coins/markets"
    let headers: HTTPHeaders = [
        "Content-Type": "application/json"
    ]
    let queryParams: Parameters = [
        "vs_currency": "usd",
        "ids": ids.lowercased()
    ]
    return apiRequest(
        .coinGecko,
        .get,
        path,
        headers,
        queryParams, nil,
        .default)
    .map { try decode([Crypto].self, from: $0.1) }
    .catch {
        parseError($0)
        return Observable.just([])
    }
}
```

Listing 14. Function, which utilizes the initial request builder. It returns an observable collection of cryptocurrency items.

New request functions presented in Listing 14, are created within the `ApiService` class, and are exposed as public functions. The class represents a Model, which contains different functions to generate data from network requests. Each function feeds data into a ViewModel, which are bound to perform layout updates.

4.2 Features

4.2.1 Chatting

To power the real time chatting feature, a server was created with Node.js. The server exposes two endpoints for the clients to utilize. Whenever a user connects to the chat, a connect endpoint is requested and the event is passed to Pusher. Lastly, when a user sends a message from the client, a message endpoint is requested, and the server respectively passes an event to Pusher.

```
const pusher = new Pusher({
  appId: process.env.APP_ID,
  key: process.env.KEY,
  secret: process.env.SECRET,
  cluster: process.env.CLUSTER,
  useTLS: process.env.USE_TLS
})

app.post('/connect', function(req, res) {
  const message = {
    name: req.query.name
  }
  pusher.trigger('chatroom', 'user_joined', message)
  res.json({success: 200})
})

app.post('/messages', function(req, res) {
  const message = {
    text: req.query.text,
    name: req.query.name
  }
  pusher.trigger('chatroom', 'new_message', message)
  res.json({success: 200})
})
```

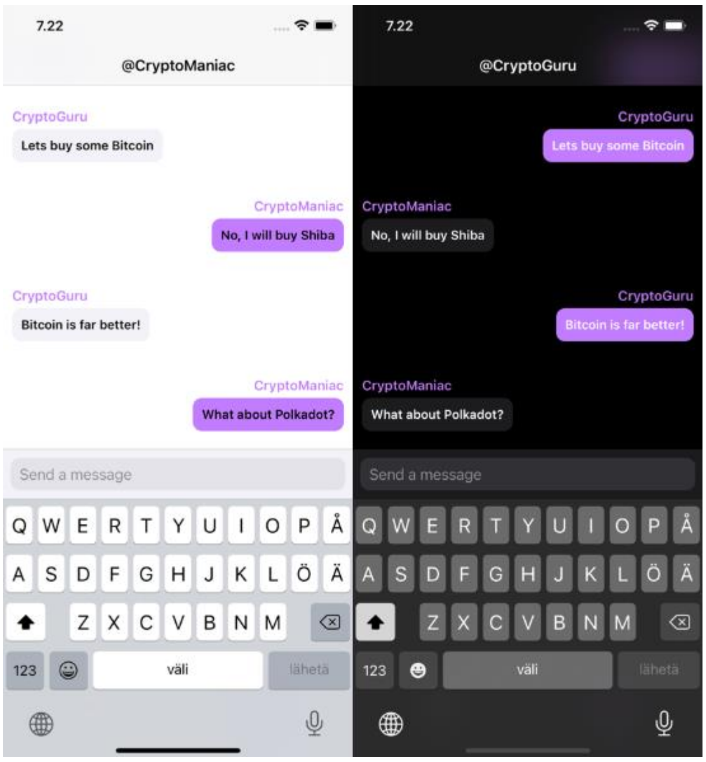
Listing 15. Node.js server implementation of Pusher.

The server-side Pusher implementation and exposed endpoints are presented in Listing 15. Pusher holds a reference to a channel which must be included in both the client and the server. Performing a HTTP request from the client to the server endpoint triggers a function within Pusher, which forwards events back to the client using a channel. The `AppDelegate` class implements the needed delegate, which is presented in Listing 16, provided from PusherSwift and subscribes to two channels.

```
AppDelegate.pusher.delegate = self
// New message
let _ = AppDelegate.channel.bind(
    eventName: "new_message",
    eventCallback: { [weak self] (event: PusherEvent) in
    if let data = event.data {
        self?.messageService.parseMessage(data: data, type: .other)
    }
})
// User joined
let _ = AppDelegate.channel.bind(
    eventName: "user_joined",
    eventCallback: { [weak self] (event: PusherEvent) in
    if let data = event.data {
        self?.messageService.parseMessage(data: data, type: .other)
    }
})
```

Listing 16. Client-side implementation of PusherSwift within AppDelegate.

Each channel can contain multiple events, which are observed in the `AppDelegate` class and handled in custom closures. The closures pass data into a `MessageService` class, which acts as a Model and notifies a ViewModel of new events. Lastly, the ViewModel outputs data to a View, which inserts new messages into the chat view and performs a layout update.



CryptoGuru
Lets buy some Bitcoin
CryptoManiac
No, I will buy Shiba
CryptoGuru
Bitcoin is far better!
CryptoManiac
What about Polkadot?

Figure 8. Real time chat view with both appearance variants.

The real time chat user interface is presented in Figure 8. The appearance is based on the device’s selection and uses default iOS colors, excluding the highlighted colors. New messages that cause the layout to update also scrolls the view to the latest message. The visible keyboard dynamically alters the layout properties of the scroll view to fit the latest message and the keyboard simultaneously on the screen.

4.2.2 Assets

The asset list presents hundred cryptocurrency items provided from the CoinGecko API. The assets are sorted by market cap rank in descending order. Each item is decoded to a `Struct`, which is presented in Listing 17. Since decoding can throw an error, each object property is treated as a nullable value to prevent fatal decoding errors. As Swift is considered as a type safe programming language, the struct exposes parsed variables that only contain valid values to minimize nil values within the code.

```
struct Crypto: Decodable {
    let id: String?
    let name: String?
    let image: String?
    let symbol: String?
    let current_price: Double?
    let market_cap: Double?
    let market_cap_rank: Int?
    let high_24h: Double?
    let low_24h: Double?
    let price_change_24h: Double?
    let price_change_percentage_24h: Double?
    let last_updated: String?
```

Listing 17. Crypto struct which represents a single cryptocurrency asset.

The initial controller, `HomeViewController`, which is presented in Listing 18, initializes the cryptocurrency list view and ViewModel The controller passes two observables into the ViewModel, the `UISearchBar` text and a `UIRefreshControl` event. The text is used to perform HTTP request to fetch specific cryptocurrency items and the refresh control event allows refreshing of the list.

```
init() {

self.cryptoListView = CryptoListView()
self.viewModel = HomeViewModel(
    searchText: cryptoListView.searchBar.rx.text.orEmpty
        .throttle(.seconds(1),
        scheduler: MainScheduler.instance)
        .distinctUntilChanged().startWith(""),
    refresh: cryptoListView.refresher.rx.controlEvent(.valueChanged)
        .asObservable().startWith(())
        .throttle(.seconds(1), scheduler: MainScheduler.instance))
super.init(nibName: nil, bundle: nil)
}
```

Listing 18. HomeViewController class initialization function.

The usage of `Observable` binding and SnapKit view constraining is presented in Listing 19. All `Disposable` objects, which are generated by the `Binder` class, are inserted in to the `DisposeBag` object for proper deinitialization via convenience function found from the `DisposeBag` class. The cryptocurrency list view is constrained to the edges of the screen with the concise SnapKit function starting on line 14. The entire listing is executed in the `viewDidLoad` function within the `HomeViewController` class.

```
1  disposeBag.insert {
2      viewModel.cryptoListLoading
3          .bind(to: animating)
4      viewModel.initialLoading
5          .bind(to: initialLoading)
6      viewModel.cryptoList
7          .bind(to: createCryptoCards)
8      viewDidAppear
9          .bind(to: setTitle)
10     viewWillAppear
11         .bind(to: refresh)
12 }
13
14 cryptoListView.snp.makeConstraints { make in
15     make.edges.equalToSuperview()
16 }
```

Listing 19. Observable binding and SnapKit constraining within the HomeViewController class.

Lifecycle observables and a `DisposeBag` object is provided from `RxViewController`, which is presented in Listing 20. The class is a custom implementation of `UIViewController`, which holds subjects that generate events for lifecycle events. Gathering lifecycle logic and business logic into a single closure within a class, allows for concise and understandable approach to subscribing and binding.

```swift
class RxViewController: UIViewController {
    let disposeBag = DisposeBag()
    private let didAppear = PublishSubject<Void>()
    private let didDisappear = PublishSubject<Void>()

    var viewDidAppear: Observable<Void> {
        return didAppear.asObservable()
    }
    var viewDidDisappear: Observable<Void> {
        return didDisappear.asObservable()
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)
        didAppear.onNext(())
    }
    override func viewDidDisappear(_ animated: Bool) {
        super.viewDidDisappear(animated)
        didDisappear.onNext(())
    }
}
```

Listing 20. Observable sequences of lifecycle events within the RxViewController class.

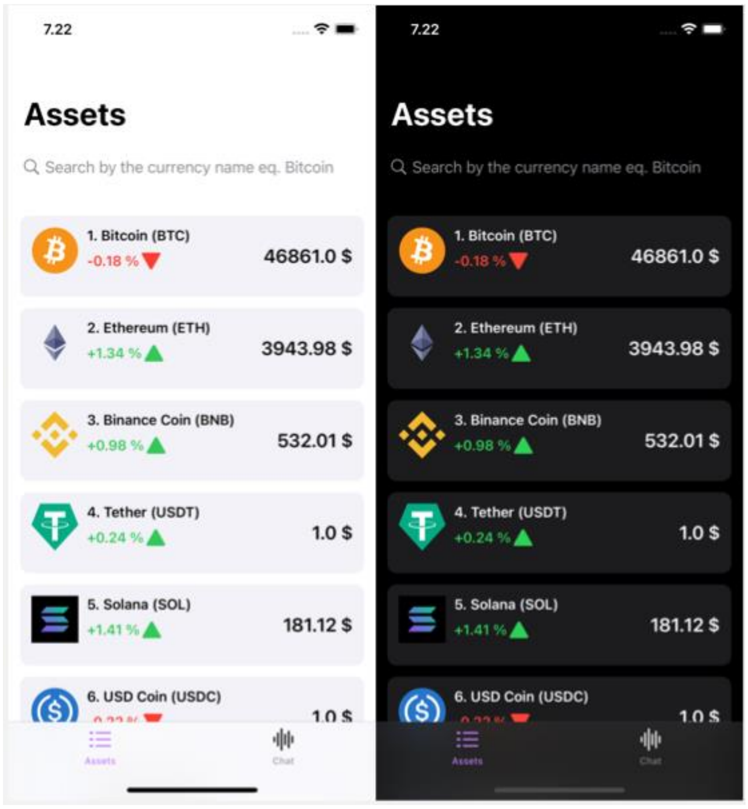
Lastly, the `HomeViewModel` class, which is presented in Listing 21, holds the business logic, and exposes observables for the `HomeViewController` to bind. `Boolean` observables are bound to display activity indicators and to hide rest of the view when data is being loaded. The collection of `Crypto` objects is bound to create the list view.

To minimize the risk of creating strong reference cycles or repeating data sequences, subscriptions should only exist within the Controller and observables only on ViewModels. The share operator, which is used on Listing 21, line 20, returns an observable sequence which shares a single subscription to the underlying source. This enables the result to be safely used to create other observables sequences within the ViewModel, while keeping a singular connection between the source and preventing multiple subscriptions to the same sequence.

```
1  class HomeViewModel {
2      let cryptoList: Observable<[Crypto]>
3      let initialLoading: Observable<Bool>
4      let cryptoListLoading: Observable<Bool>
5
6  init(
7      searchText: Observable<String>,
8      refresh: Observable<Void>
9      ) {
10     let refreshAndText = Observable.combineLatest(
11         searchText, refresh
12     )
13
14     cryptoList = refreshAndText
15         .flatMapLatest {
16             ApiService.instance.getCryptoCurrencies(
17                 ids: $0.0
18             )
19         }
20         .share(replay: 1, scope: .forever)
21         .startWith([])
22
23     cryptoListLoading = Observable.merge(
24         cryptoList.map {
25             $0.count == 0
26         }.startWith(false),
27         refresh.mapTo(constant: true)
28     ).startWith(true)
29
30     initialLoading = Observable.merge(
31         cryptoList.mapTo(constant: false).take(2),
32         cryptoList.map {
33             $0.count == 0
34         }.take(2).startWith(true)
35     ).startWith(true)
36     }
37 }
```

Listing 21. Business logic and bindable observables within the HomeViewModel class.

Furthermore, the take operator, which is used on Listing 21, line 31, ensures that only specific number of contiguous elements, starting from the first, are included in the sequence. After the sequence has produced the specific number of elements, it completes, and the underlying source is disposed of.



7.22

Assets

Search by the currency name eq. Bitcoin

1. Bitcoin (BTC) -0.18 % ▼ 46861.0 $
2. Ethereum (ETH) +1.34 % ▲ 3943.98 $
3. Binance Coin (BNB) +0.98 % ▲ 532.01 $
4. Tether (USDT) +0.24 % ▲ 1.0 $
5. Solana (SOL) +1.41 % ▲ 181.12 $
6. USD Coin (USDC) 1.0 $

7.22

Assets

Search by the currency name eq. Bitcoin

1. Bitcoin (BTC) -0.18 % ▼ 46861.0 $
2. Ethereum (ETH) +1.34 % ▲ 3943.98 $
3. Binance Coin (BNB) +0.98 % ▲ 532.01 $
4. Tether (USDT) +0.24 % ▲ 1.0 $
5. Solana (SOL) +1.41 % ▲ 181.12 $
6. USD Coin (USDC) 1.0 $

Figure 9. Cryptocurrency asset list view with both appearance variants.

The cryptocurrency asset list view is presented in Figure 9. Typing on the search bar or performing a pull-down gesture on the view, will cause the list to refresh and regenerate the list items. The search bar text is throttled to only include distinct values between a time period to save resources and to prevent fetching irrelevant data. Each list item holds a subscription to display a detailed view of a single asset.

4.2.3 Details

The second level navigation presents a detailed view of a single cryptocurrency asset. Each view contains a graph, which displays historical price data of an asset. The graph is created with Charts, which is a pure Swift library for iOS, tvOS and macOS allowing to create graphical representations from numerical data. [27]

Data for the graph is provided from CoinGecko API and is decoded into a collection of `Double` arrays. The collection contains two values, timestamp of the price in Unix format and the price. Timestamp is interpreted either in days, hours or minutes depending on the query parameter of the HTTP request. [28]

```
1  let marketChart = daysAndId
2      .flatMapLatest {
3          ApiService.instance.getChartData(
4              id: $0.0,
5              days: $0.1
6          )
7      }
8      .share(replay: 1, scope: .forever)
9
10 let yValues = marketChart
11     .filter { $0.prices.count > 0 }
12     .flatMap { chart -> Observable<[ChartDataEntry]> in
13         return Observable.just(chart.prices.map {
14             ChartDataEntry(
15                 x: $0[0],
16                 y: $0[1].rounded(toPlaces: 3)
17              )
18         })
19     }.share(replay: 1, scope: .forever)
20
21 marketChartData = Observable.combineLatest(
22     yValues, cryptoName, dayStream
23 )
```

Listing 22. Business logic for market chart data within the DetailsViewModel class.

The business logic for displaying market chart data is presented in Listing 22. The decoded data provided from the HTTP request is converted into a collection of `ChartDataEntry`, which is a class provided from Charts. The class contains entities for x- and y-axis values and represents a single entry in a chart.

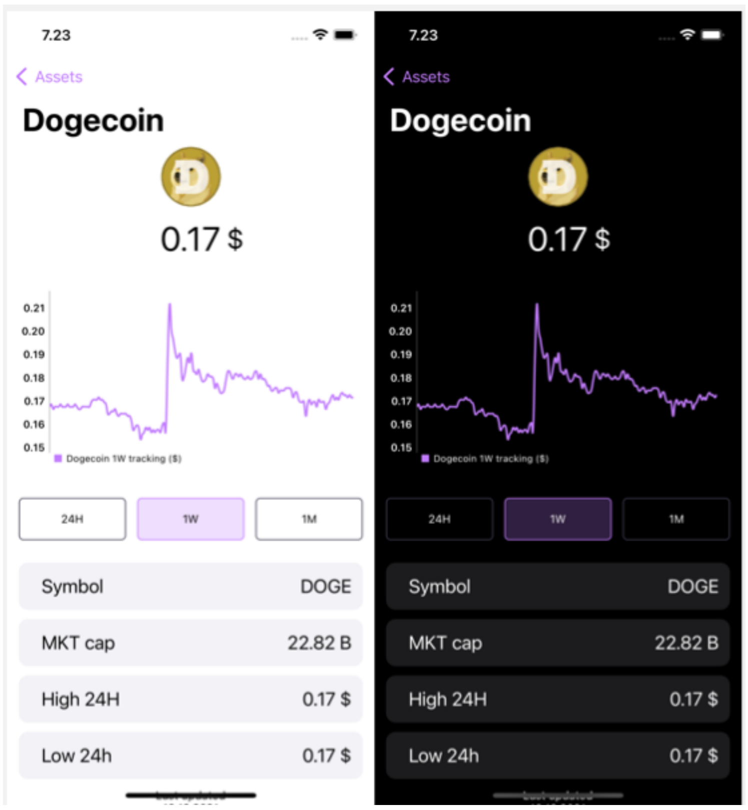
By utilizing the combine operator, the resulting observable sequence is of type `([ChartDataEntry], String, MarketDays)`, where the `String` represents the cryptocurrency name and the `MarketDays`, the time interval of the chart entries in an enumerated format. The share operators on Listing 22, line 45 and line 56, ensure that the sequences are safe to be combined without performing any additional resource wasting networking requests or generating multiple subscriptions to the source observable.

```
1  var setData: Binder<(
2      [ChartDataEntry], String, MarketDays
3  )> {
4      return Binder(self) { owner, data in
5          let label: String = {
6              switch data.2 {
7              case .week:
8                  return "\(data.1) 1W tracking ($)"
9              case .day:
10                 return "\(data.1) 24H-tracking ($)"
11             case .month:
12                 return "\(data.1) 1M-tracking ($)"
13             }
14         }()
15         let set1 = LineChartDataSet(
16             entries: data.0, label: label
17         )
18         let data = LineChartData(dataSet: set1)
19         set1.mode = .cubicBezier
20         set1.lineWidth = 2
21         set1.setColor(UIColor.Cluster.instance.action)
22         set1.drawCirclesEnabled = false
23         data.setDrawValues(false)
24         owner.detailsView.chartView.data = data
25     }
26 }
```

Listing 23. Binder within the DetailsViewController class.

The binding of the market chart data is presented in Listing 23. The custom `Binder`, which exists within the `DetailsViewController` class, acts as a bindable sink for observable sequences and sets all necessary data in to the chart view and performs a layout update. By default, binders perform on the main scheduler and cannot receive errors.

Furthermore, binders will always include two parameters in the builder function, the retainer, and the value of the observable sequence. The retainer is captured during the initialization and can be treated safely afterwards. On Listing 23, line 4, the `DetailsViewController` is retained with the `self` parameter, which refers to an implicit property found from every object in Swift. This reduces the risk of creating strong reference cycles, since `self` should be treated weakly within subscriptions.



Symbol	DOGE
MKT cap	22.82 B
High 24H	0.17 $
Low 24h	0.17 $

Symbol	DOGE
MKT cap	22.82 B
High 24H	0.17 $
Low 24h	0.17 $

Figure 10. Detailed view of a cryptocurrency asset with both appearance variants.

The detailed view of a cryptocurrency asset is presented in Figure 10. All the currency data is shown in U.S. Dollars. In addition to the chart view, the view displays basic information of the cryptocurrency asset such as the current price, market cap and the highest and lowest price within the last 24 hours. Tapping any of the selectors found from the graph view regenerates the graph with different data.

5 Conclusion

Becoming proficient in Rx library enables developers to build applications for various platforms with little effort. For mobile developers, the ability to develop applications for both iOS and Android are considered as an asset. Reactive applications are robust and agile regarding their user experience. Data binding allows for the user interface to always represent its latest state. By utilizing operators, creating complicated application logic, and transforming reactive sequences becomes straightforward.

Building user interface with UIKit solves numerous issues regarding iOS storyboards. Handling navigation and lifecycle methods programmatically, enable clean and maintainable code. Providing AutoLayout capabilities with SnapKit allows for supporting multiple sized device idioms.

Since no programming paradigm is perfect, even RxSwift includes issues. They might regard certain behavior within operators and the generated sequences. Debugging unwanted side effects or misbehaving data streams can be a tedious process. Creating system wide models, such as caches or WebSocket repositories with shared observables can cause repeating data sequences. Strong reference cycles and memory leaks might cause cumulatively increasing subscriptions if the sequences are not disposed properly.

All the problems combined with the high learning curve of RxSwift can be challenging for unskilled iOS developers. Since UIKit lacks the ability to inspect view components, building layout with UIKit requires sufficient knowledge of user interface components and navigation hierarchy.

MVVM combined with RxSwift and RxCocoa offers a future proof way of building iOS applications. Since Apple’s own reactive framework, Combine, and RxSwift share similar ideology, building applications with RxSwift enable migration to Combine in the future. Numerous RxSwift contributors are creating a bi-directional framework, RxCombine, to bridge Combine and RxSwift together to provide more stable migration tools, or even codevelop using both frameworks.

References

1 Pillet, Florent; Todorov, Marin; Mishali, Shai; Gardner, Scott & Bontognali, Junior. 2017. RxSwift: Reactive Programming with Swift. 4th ed. Razeware.

2 Apple Developer. Combine. Online. <https://developer.apple.com/documentation/combine>. Accessed 22.11.2021.

3 GitHub. RxSwift. Online. <https://github.com/ReactiveX/RxSwift>. Accessed 22.11.2021.

4 Raywenderlich Tutorial Team & Sullivan, Alex. 2019. Reactive Programming with Kotlin: Learn Rx with RxJava, RxKotlin, and RxAndroid. 1st ed. Razerware.

5 ReactiveX. Online. <https://reactivex.io>. Accessed 23.11.2021.

6 Mishali, Shai. 2018. RxSwift: debunking the myth of hard. Online. <https://www.youtube.com/watch?v=-N2qtTnP1sg>. Accessed 27.11.2021.

7 Signh, Vaibhav. 2021. RxSwift: Finite and Infinte Observables. Online. <https://vaibhavsingh-54243.medium.com/rxswift-finite-and-infinite-observables-69e4e4469be7>. Accessed 28.11.2021.

8 The Swift Programming Language. 2021. Automatic Reference Counting. Online. <https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html>. Accessed 28.11.2021.

9 Raywenderlich.com. RxSwift in Practice - RWDevCon 2016 Live Tutorial Session. 2017. Online. <https://youtu.be/W3zGx4TUaCE>. Accessed 28.11.2021.

10 ReactiveX. Subject. Online. <https://reactivex.io/documentation/subject.html>. Accessed 28.11.2021.

11 Gümüs, Göktug. 2020. Getting started with RxSwift and RxCocoa. Online. <https://medium.com/flawless-app-stories/getting-started-with-rxswift-and-rxcocoa-5534cf2902b7>. Accessed 4.12.2021

12 Pandey, Yuvraj. 2018. RxSwift: Observing Operators. Online. <https://yuvrajpy.medium.com/rxswift-observing-operators-c54b46a9a778>. Accessed 4.12.2021

13 ReactiveX. Operators. Online <https://reactivex.io/documentation/operators.html>. Accessed 6.12.2021

14 Mróz, Lukasz. 2016. RxSwift by Examples #4 – Multithreading. Online. <https://www.thedroidsonroids.com/blog/rxswift-examples-4-multithreading>. Accessed 6.12.2021

15 Campbell, Lee. 2015. Introduction to Rx: Scheduling and Threading. Online. <https://introtorx.com/Content/v1.0.10621.0/15_SchedulingAndThreading.html>. Accessed 12.12.2021.

16 Campbell, Lee. 2015. Introduction to Rx: Hot and Cold Observables. Online. <http://introtorx.com/Content/v1.0.10621.0/14_HotAndColdObservables.html>. Accessed 12.12.2021.

17 Kliffer, Ron. 2019. Getting Started with RxSwift and RxCocoa. Online. <https://www.raywenderlich.com/1228891-getting-started-with-rxswift-and-rxcocoa>. Accessed 17.12.2021.

18 Suojanen, Jussi. 2018. How to use RxSwift with MVVM pattern – part 1. Online. <https://www.vincit.fi/en/rxswift-with-mvvm-part1/>. Accessed 17.12.2021.

19 Apple Developer. 2018. Model-View-Controller. Online. <https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>. Accessed 18.12.2021.

20 Law, Raymond. MVC doesn’t lend itself well to unit testing. Online. <https://clean-swift.com/mvc-doesnt-lend-well-unit-testing/>. Accessed 18.12.2021.

21 Douglas, Aaron. 2016. WebSockets on iOS with Starscream. Online. <https://www.raywenderlich.com/861-websockets-on-ios-with-starscream>. Accessed 18.12.2021.

22 Pusher. Online. <https://pusher.com/channels>. Accessed 18.12.2021.

23 Apple Developer. UIKit. Online. <https://developer.apple.com/documentation/uikit>. Accessed 19.12.2021.

24 Cherednichenko, Sveta. Pros and Cons of Working with Storyboards. Online. <https://www.mobindustry.net/blog/pros-and-cons-of-working-with-storyboards/>. Accessed 19.12.2021.

25 Mishali, Shai. 2019. SnapKit for iOS: Constraints in a Snap. Online. <https://www.raywenderlich.com/3225401-snapkit-for-ios-constraints-in-a-snap>. Accessed 19.12.2021.

26 CocoaPods. Online <https://cocoapods.org/about>. Accessed 23.12.2021.

27 Gindi, Daniel. 2020. Charts. Online. <https://github.com/danielgindi/Charts>. Accessed 6.1.2022.

28 CoinGecko. 2021. CoinGecko API documentation. Online. <https://www.coingecko.com/en/api/documentation>. Accessed 6.1.2022.