Combine for RxSwift Developers (quickstart/guide)

Intro

This guide is a practical introduction to `Combine` for developers who have experience with `RxSwift`, so that you can get up to speed quicky and start writing production code as quickly as possible. The goal is accelerated learning, so that you can start writing production code with `Combine` in no time at all. While most of the knowledge you have about `RxSwift` translates well to `Combine`, going through existing resources, tutorials, books and blog posts can be time-consuming. This guide aims to summarize what you really need to know to get started and provide a jumping platform for more in-depth learning. You can treat it as a quick reference.

Additionaly, a nicely formatted printable PDF version of this guide is available *here*.

Problem

If you're reading this, chances are that your codebase relies heavily on `RxSwift` streams. You're probably using them with `UIKit`. Now, you might be considering introducing new `SwiftUI` components, but there's a catch - `SwiftUI` bindings use `Combine`, not `RxSwift`. So, how can you feed your new Views with data from the existing streams?

The good news is, you don't need to rewrite existing RxStreams into `Combine`. Instead, all you need is a translation layer between the two. Thankfully, this is a fairly common problem and it's been solved by an open source library: RxCombine. It provides bi-directional bindings and they're extremely easy to work with.

It's most likely not worth rewriting existing `RxSwift` streams into `Combine`. Instead, you could plug your existing streams into `SwiftUI` components using a translation layer.

This can be achieved with `RxCombine` library (https://github.com/CombineCommunity/RxCombine). Bi-directional bindings are provided and they are extremely easy to work with.

Question: is it worth rewriting existing `RxSwift` streams into Combine?

In most cases, it's probably not worth the effort to manually rewrite them into `Combine`. Here are a few reasons why:

- The existing code is already working and has likely been thoroughly tested. Rewriting it could introduce new issues that would take time to debug and fix.
- `RxSwift` code is highly reusable, ReactiveX framework has been implemented in multiple programming languages. It's possible to reuse your `RxSwift` code across different platforms with minimal modifications.
- Depnding on the size and complexity of the codebase, it may take a significatn amount of time and effort to rewrite everything in `Combine`, with no guarantee of significant improvement in performance or functionality.

`RxSwift` vs Combine: Basic things to consider/remember

- `RxSwift` has `RxCocoa` bindings, which make it eaqsy to work with `UIKit`.
- While `Combine` has some native `UIKit` bindings, they are fairly basic. More extensive implementation is provided via a 3rd party library `CombineCocoa`, but it's far from complete.
- `RxSwift` is compatible with both Swift and `Objective-C`, but `Combine` only works with Swift.
- `Combine` is available on iOS13 and later, but some `SwiftUI` features are only available on iOS 14 or 15. If your code targets iOS 13 or 14, you might need to use backports or conditional execution to implement some functionality. For example good Markdown support in `SwiftUI` is only available on iOS 15 or later.

Further reading:

- *Hacking with Swift: What's new in SwiftUI for iOS15* (https://www.hackingwithswift.com/articles/235/whats-new-in-swiftui-for-ios-15)
- *MongoDB blog: Most Useful iOS15 SwiftUI Features* (https://www.mongodb.com/developer/products/realm/realm-ios15-swiftui/).

Terminology differences

- `Observables` vs. `Publishers`: In `RxSwift`, a stream of values is called an `Observable`, while in Swift `Combine`, it's called a `Publisher`.

- `Observers` vs. `Subscribers`: In `RxSwift`, an object that receives values from an `Observable` is called an `Observer`, while in Swift `Combine`, it's called a `Subscriber`.
- Both `RxSwift` and Swift `Combine` use operators to transform and `Combine` streams of values.

Syntax

Syntax between `RxSwift` and `Combine` is quite similar, however `Combine Publishers` use associated types instead of generics, which makes them strongly typed. In practice this means you'll call `eraseToAnyPublisher()` *A LOT* to make your code more practical and concise.

Creating and subscribing to simple streams (publishers)

Creating streams (publishers)

In `RxSwift` simple streams can be created using the `Observable.create` function or by using predefined methods like `just`, `from`, or `of`.

In `Combine` they're called `Publishers` and they can be created using predefined methods like `Just`, `Publishers.Sequence`.

Example:

```
let combinePublisher = Just("Hello world!")
```

Subscribing to publishers

Subscribing to a `Combine` publisher using the `sink` method.

Example:

```
combinePublisher.sink(receiveCompletion: { completion in
    switch completion {
    case .finished:
        print("Completed")
    case .failure(let error):
        print("Error: \(error)")
    }
}, receiveValue: { value in
    print(value)
})
```

Memory management (cancelling subscriptions)

In both technologies, you still have to remember to dispose your streams to avoid memory leaks.

- `DisposeBag` vs. `AnyCancellable`: In `RxSwift`, the `DisposeBag` class is used to manage subscriptions and ensure that they are cancelled when the object that owns them is deallocated. In Swift `Combine`, the `AnyCancellable` class is used for the same purpose.
- Cancelling subscriptions: In both frameworks, subscriptions can be cancelled manually using the `dispose` method (in RxSwift) or the `cancel` method (in Combine).
- `Combine sink` operator (`subscribe` equivalent) creates a subscription that needs to be taken care of.
- You can use `store(in:)` method to bind a subscription to a set of `AnyCancellables` (effectively mimicking `RxSwift DisposeBag`)

Here's how this looks in practice in `Combine`:

```
import Combine

class DataFetchService {
    fileprivate var cancellables = Set<AnyCancellable>()
```

```
    func fetchData() {
        URLSession.shared.dataTaskPublisher(for: URL(string: "https://example.com")!)
            .map(\.data)
            .decode(type: MyData.self, decoder: JSONDecoder())
            .sink(receiveCompletion: { _ in }, receiveValue: { data in
                // Handle the fetched data here
            })
            .store(in: &cancellables) // Add the cancellable to the set
    }
}
```

When the `DataFetchService` instance gets out of scope and is deallocated, subscription will be removed.

You could add the following method for allowing manual disposal:

```
extension DataFetchService {
    /**
     * If you don't call this manually, the subscription above
     * will be removed when the `DataFetchService` object is removed.
     */
    func cancelFetch() {
        cancellables.forEach(\.cancel)
        cancellables.removeAll()
    }
}
```

As an additional note, it's generally recommended to place `cancellables` in your view controller when using `UIKit` in `Combine`. This is because view controllers have a well-defined lifecycle, unlike `ViewModels`, and this ensures proper cancellation and prevents leask.

Example:

```
struct PlaneViewModel {
    private let plane = Plane()

    let currentAltitudeText: AnyPublisher<String, Never>

    init() {
        currentAltitudeText = plane.$altitude.map { "\($0) feet" }.eraseToAnyPublisher()
    }
}

class FlightDashboardViewController {
    let viewModel = PlaneViewModel()
    let label = UILabel()

    var cancellables = Set<AnyCancelable>()

    override func viewDidLoad() {
        super.viewDidLoad()
        createSubscriptions()
    }

    private func createSubscriptions() {
        viewModel.currentAltitudeText
            .assign(to: \.text, on: label)
            .store(in: &cancellables)
    }
```

```
}
```

Operator naming differences

- `reduce` (RxSwift) -> `scan` (Combine)
- `flatMapLatest` (RxSwift) -> `switchToLatest` Switch marble diagram explanation
- It's worth reading on the `scan` and `switchToLatest` behaviour as they're similar, but not exactly the same as their `RxSwift` equivalents (detailed explanation is provided later).
- Operators such as `map`, `filter`, `flatMap`, `merge`, `zip`, `combineLatest`, `debounce`, `throttle` have the same name in both frameworks.

Example:

```
let numbers = Publishers.Sequence(sequence: [1, 2, 3])

numbers.filter { $0 % 2 == 0 }
    .map { "\($0) is even" }
    .sink(receiveValue: { value in
        print(value)
    }).store(in: &cancellables)
```

More on `flatMapLatest` vs `switchToLatest`

When nesting publishers, `switchToLatest` only subscribes to the publisher that emited the last value, cancelling subscription to other publishers. This is best explained with an example:

```
import Combine

let nestedPublishers = PassthroughSubject<PassthroughSubject<Int, Never>, Never>()

let subscription = nestedPublishers
    .switchToLatest()
    .sink { value in
        print("Received value: \(value)")
    }

let publisher1 = PassthroughSubject<Int, Never>()
let publisher2 = PassthroughSubject<Int, Never>()

nestedPublishers.send(publisher1)
publisher1.send(1)
publisher1.send(2)

nestedPublishers.send(publisher2)
publisher1.send(3) // Will not be received
publisher2.send(4)
```

Note that if this was `RxSwift` and `flatMapLatest`, the value '3' would still be printed out.

Publisher.CombineLatestX

Some of the operators have a variant implemented on a generic `Publisher` type that contain number of expected parameters, for example: `Publishers.CombineLatest` (supports only 2 publishers), `Publishers.CombineLatest3`, `Publishers.CombineLatest4`.

This can make the syntax rather awkward when you want to use `combineLatest` with more than 4 streams, since there are no variants of the operator with more than four parameters (such as `CombineLatest5`):

```
Publishers.CombineLatest4($var0, $var1, $var2, $var3)
    .combineLatest($var4)
```

```
    .combineLatest($var5)
    .sink { completion
        // ...
    }.receiveValue: { res0, res1 in
        let variables = res0.0
        let var0 = variables.0
        let var1 = variables.1
        let var2 = variables.2
        let var3 = variables.3
        let var4 = res0.1
        let var5 = res1
        // ...
    }.store(in: &cancellables)
```

The method illustrated above also works for `Publishers.Zip3`, `Publishers.Zip4`, etc.

Note that there's an alternative way, but it makes the syntax even more cumbersome:

```
let cancellable = publisher1.combineLatest(publisher2)
    .combineLatest(publisher3)
    .combineLatest(publisher4)
    .combineLatest(publisher5)
    .sink { res in
        let val1 = res.0.0.0.0
        let val2 = res.0.0.0.1
        let val3 = res.0.0.1
        let val4 = res.0.1
        let val5 = res.1

        print("Combined values: \(val1), \(val2), \(val3), \(val4), \(val5)")
    }
```

If you find yourself hitting this scenario often, it's worth considering extending Publisher with a generic combineLatest function variant supporting more parameters.

If you frequently encounter this scenario, you might want to consider extending the `Publisher` type with a generic function variants that support more parameters for `combineLatest`.

In `CombineExt library`, `Collection.combineLatest` function solves that problem. Example usage: `[Just(1), Just(2), Just(3), Just(4)].combineLatest()`.

Futures

`Future` is an equivalent of an `RxSwift Single`. It emits value or error once (based on a provided closure). This is used for performing an async operation and returning a result. You create a `Future` by providing a `Promise` using a closure:

```
import Combine

struct DataFetchError: Error {}

func fetchData() -> AnyPublisher<String, Error> {
    return Future<String, Error> { promise in
        DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
            let success = Bool.random()
            if success {
                promise(.success("Data fetch successful!"))
            } else {
                promise(.failure(DataFetchError()))
            }
```

```
        }
    }.eraseToAnyPublisher()
}

let fallbackPublisher = Just("fallback").setFailureType(to: Error.self)

let subscription = fetchData()
    .catch { _ in fallbackPublisher }
    .sink(
        receiveCompletion: { print($0) },
        receiveValue: { print($0) }
    )

subscription.cancel()
```

Subjects

PublishSubject -> PassthroughSubject

`PassthroughSubject` is an equivalent of a `PublishSubject`. It allows you to manually emit values and bridge the gap between non-reactive and `Combine` code.

Example:

```
import Combine

let subject = PassthroughSubject<String, Error>()

subject.sink(
    receiveCompletion: { print($0) },
    receiveValue: { print($0) }
).store(in: &cancellables)

subject.send("Hello, world!")
```

BehaviorSubject -> CurrentValueSubject

`CurrentValueSubject` works the same way in `Combine` as `BehaviorSubject` in RxSwift.

```
import Combine

let currentValueSubject = CurrentValueSubject<Int, Never>(0)

// Prints "Initial value: 0"
print("Initial value: \(currentValueSubject.value)")

let subscriber = currentValueSubject.sink { value in
    // Prints "Received value: 1", then "Received value: 2"
    print("Received value: \(value)")
}

currentValueSubject.send(1)
currentValueSubject.send(2)
```

ReplaySubject

`ReplaySubject` in `RxSwift` is similar to `BehaviorSubject`, but you can specify the length of a buffer to reply (you can think of `BehaviorSubject` as a `ReplaySubject` with a buffer length equal 1).

`ReplaySubject` is not available in `Combine` by default.

You can use `ReplaySubject` from `CombineExt` library (https://github.com/CombineCommunity/CombineExt).

Common native SDK publishers

In general the standard library has publishers where you'd expect them. Examples:

- `NotificationCenter.Publisher`
- `Timer.Publisher`
- `URLSession.DataTaskPublisher`
- `UrlSession.DownloadTaskPublisher` etc.

Operator usage differences

There's a handy cheat sheet listing `RxSwift` versions and their `Combine` equivalents: RxSwift to `Combine` Cheat Sheet.

RxCombine example (converting streams into publisheres and vice-versa)

Example of how to use RxCombine to mix `Combine` and `RxSwift` streams:

```
import Combine
import RxSwift
import RxCombine

let combinePublisher = Just(42)

let rxSwiftObservable = Observable<Int>.from([1, 2, 3])

let stream = Publishers.CombineLatest(combinePublisher, rxSwiftObservable.rx.publisher())
    .map { $0 + $1 }
    .asObservable()

_ = stream.subscribe(onNext: { print($0) })
```

Threading and Schedulers

- `observeOn` vs. `receive(on:)`: In `RxSwift`, the `observeOn` operator is used to specify the scheduler on which to receive events, while in Swift `Combine`, the `receive(on:)` operator is used for the same purpose.
- `subscribeOn`: In `RxSwift`, the `subscribeOn` operator is used to specify the scheduler on which to subscribe to an Observable. This operator is not available in Swift `Combine`.
- Available schedulers are similar, though the names differ
  - `RxSwift` builtin schedulers:
    - `CurrentThreadScheduler` (default), `MainScheduler`
    - `SerialDispatchQueueScheduler`, `ConcurrentDispatchQueueScheduler`
    - `OperationQueue` scheduler.
  - `Combine` builtin schedulers:
    - `DispatchQueue`
    - `OperationQueue`
    - `RunLoop`
    - `ImmediateScheduler`
- In `Combine` if you don't specify a scheduler explicitly, the same that generated a published value is used.

Example:

```
import Combine

let queue = DispatchQueue(label: "com.example.my-queue")

let publisher = URLSession.shared.dataTaskPublisher(for: URL(string: "https://www.example.com")!)
    .map { $0.data }
    .subscribe(on: queue)
    .decode(type: String.self, decoder: JSONDecoder())
    .receive(on: DispatchQueue.main)
    .sink(
        receiveCompletion: { print($0) },
        receiveValue: { print($0) }
    )
```

Sharing Subscriptions

`share()` operator works similarily. It's used to create a shared publisher that sends values to multiple subscribers while keeping only one connection to the original publisher. It helps to avoid repeating expensive tasks like network requests. Unlike `multicast`, `share()` handles connections automatically based on active subscribers.

Unlike `RxSwift`, there's no option to specify replay count or scope.

Example:

```
import Combine
import Foundation

let url = URL(string: "https://api.example.com/data")!
let sharedPublisher = URLSession.shared.dataTaskPublisher(for: url).share()

let subscriber1 = sharedPublisher.sink { ... }
let subscriber2 = sharedPublisher.sink { ... }
```

In this example, both subscriber1 and subscriber2 receive values from a single network request, avoiding duplication.

Backpressure

In `RxSwift` there's no implementation of backpressure. In `Combine` it allows subscribers to control the rate at which they receive values from publishers. This introduces a concept of demand. When a subscriber subscribes to a publisher, it can specify its initial demand, and it can request more values later if needed.

Example:

```
import Combine

let publisher = (1...10).publisher
let subscriber = CustomSubscriber()

publisher.subscribe(subscriber)

class CustomSubscriber: Subscriber {
    typealias Input = Int
    typealias Failure = Never

    private var subscription: Subscription?

    func receive(subscription: Subscription) {
        self.subscription = subscription
        subscription.request(.max(3))
    }
```

```
    func receive(_ input: Int) -> Subscribers.Demand {
        print("Received value: \(input)")
        return input < 3 ? .max(1) : .none
    }

    func receive(completion: Subscribers.Completion<Never>) {
        print("Completed")
    }
}
```

Debugging

In `Combine`, `print()` operator is available (equivalent to `debug()` from RxSwift.

Additionally, handleEvents allows you to add side effects (like logging or breakpoints, for specific publisher events.

```
import Combine

let numbersPublisher = [1, 2, 3, 4].publisher

let subscription = numbersPublisher
    .print("Numbers")
    .handleEvents(receiveOutput: { value in
        if value == 3 {
            print("(Breakpoint): Value == 3")
        }
    })
    .filter { $0 % 2 == 0 }
    // .map { $0 * $0 }
    .sink { print($0) }
```

Hot and cold publishers

In `Combine`, publishers can be classified into two categories: hot and cold publishers.

Hot Publishers

Hot publishers are active, meaning they produce values regardless of whether they have active subscribers. When a new subscriber subscribes to a hot publisher, it will receive only the values emitted after the subscription is made. Examples of hot publishers include system events, notifications, and subjects.

Cold Publishers

Cold publishers are inactive until they have at least one subscriber. They start producing values only when a subscriber subscribes to them, and they will replay the entire sequence of values for each new subscriber. Examples of cold publishers include `URLSession.dataTaskPublisher`, `Just`, and `Sequence.publisher`.

Error Handling

`catch`: Allows you to recover from an error by returning an alternate publisher. The error is *NOT* propagated downstream, and the new publisher continues emitting values.

```
import Combine

let publisher = PassthroughSubject<Int, Error>()

let subscription = publisher
    .catch { error -> Just<Int> in
```

```
        print("Error occurred: \(error)")
        return Just(0)
    }
    .sink { value in
        print("Received value: \(value)")
    }

publisher.send(1)
publisher.send(completion: .failure(MyError.exampleError))
publisher.send(2)
```

`retry` is also available and working as expected:

```
import Combine
import Foundation

let url = URL(string: "https://api.example.com/data")!
let urlPublisher = URLSession.shared.dataTaskPublisher(for: url)

let subscription = urlPublisher
    .retry(3)
    .sink { completion in
        print(completion)
    } receiveValue: { data, response in
        print("Received data: \(data)")
    }
```

`replaceError` is also available.

Unit Tests

Use `TestPublisher` and `TestSubscriber` provided by the `CombineXCTest` package to test your custom publishers and subscribers.

For testing asynchronous code, use `XCTestExpectation`, which allows you to wait for certain conditions to be met before proceeding with the test.

To test operators that introduce delays or time-based operations, like `debounce` or `throttle`, you can use a `TestScheduler` to control the passage of time during tests.

Useful libraries: CombineExt

CombineExt library (https://github.com/CombineCommunity/CombineExt) provides implementation of some operators common in `ReactiveX` that are not implemented in `Combine`. This makes the transition a bit easier when writing new publishers.

Example operators provided:

- `withLatestFrom`
- `amb` (takes multiple publishers and only emits a value of the first one to emit)
- `flatMapLatest`
- `Collection.combineLatest` (example: `[Just(1), Just(2), Just(3), Just(4)].combineLatest()`)
- `Publisher.share(replay:)` (same as `Publisher.share` but lets you specify buffer length - number of last elements to reply)
- `PassthroughRelay` (RxSwift `PublishRelay` equivalent)

Summary

This should provide you with enough information to start working with `Combine` publishers (assuming `RxSwift` knowledge). If you spot any errors in the guide, don't hesitate to contact me via email/*LinkedIn* (https://www.linkedin.com/in/tomaszwyszomirski/).