Rx(Swift), FRP and everything in between

What is FRP

- Functional Reactive Programming
- Comes from the Haskell world (Of course it does 🙄)
- Idea of representing a view(they call it a widget in the paper)'s content at any time as a "behaviour" - a time varying value.
- Rather than setting a value as a side effect, represent as continuously changing values over time.

Lots of implementations of FRP-like programs

- RxSwift
- RAC/ReactiveSwift(usually would be my first choice but the documentation is 😞)
- Interstellar
- many more

Result types in Interstellar vs RxSwift

- RxSwift has error type

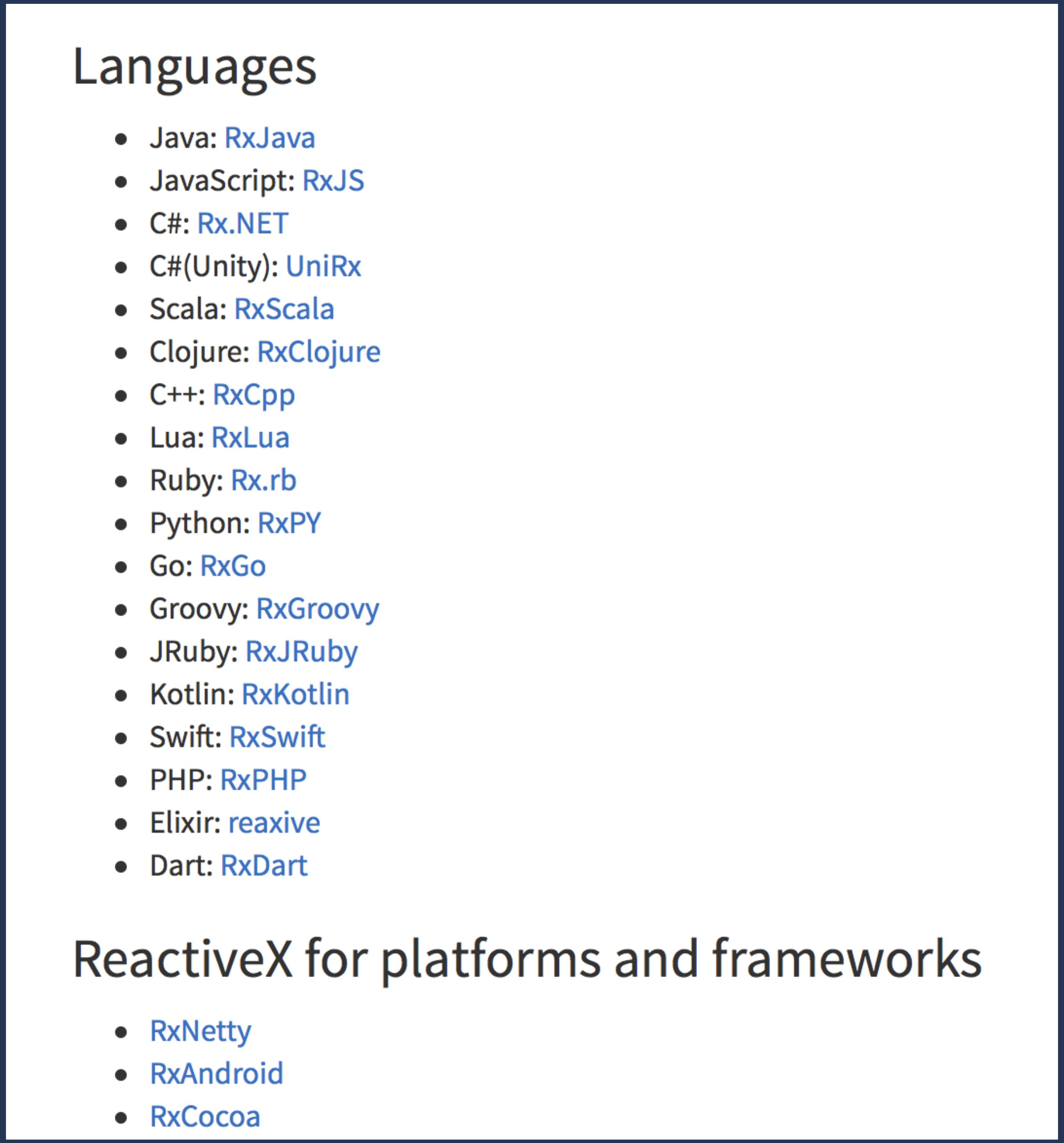
```
Result<T, Error: Swift.Error>
```

- Interstellar is BYOR (Bring Your Own Result™) + observer (as long as it follows a protocol)

What is Rx?

- Rx is functional
- Rx is reactive
- Rx is programming (obviously 😏)
- But Rx isn't FRP. Main difference is - continuous vs discrete values. My hypothesis - rejigging your UI with continuous inputs is a bad idea.

- Rx is cross platform



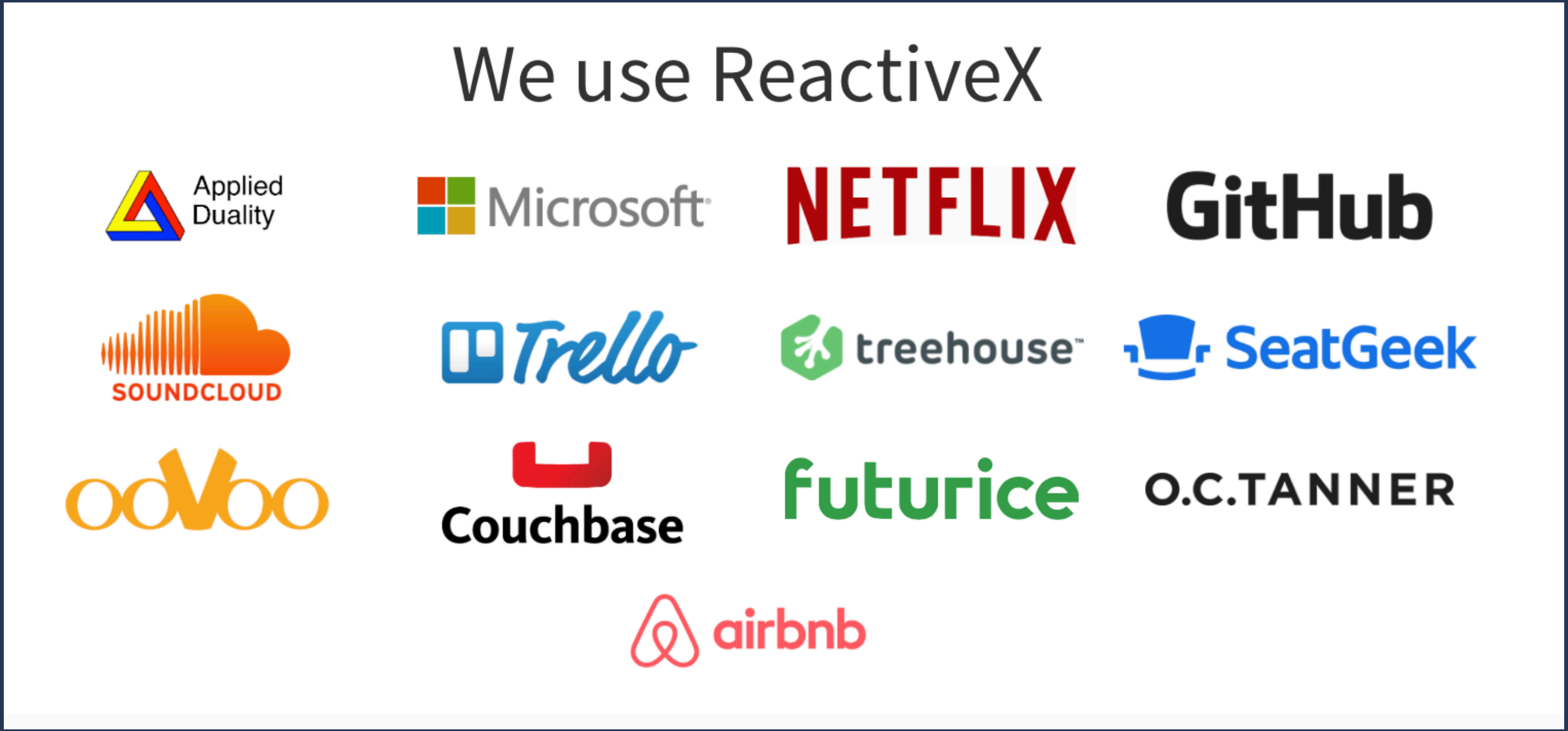
Languages

- Java: RxJava
- JavaScript: RxJS
- C#: Rx.NET
- C#(Unity): UniRx
- Scala: RxScala
- Clojure: RxClojure
- C++: RxCpp
- Lua: RxLua
- Ruby: Rx.rb
- Python: RxPY
- Go: RxGo
- Groovy: RxGroovy
- JRuby: RxJRuby
- Kotlin: RxKotlin
- Swift: RxSwift
- PHP: RxPHP
- Elixir: reaxive
- Dart: RxDart

ReactiveX for platforms and frameworks

- RxNetty
- RxAndroid
- RxCocoa

- Rx is everywhere



We use ReactiveX

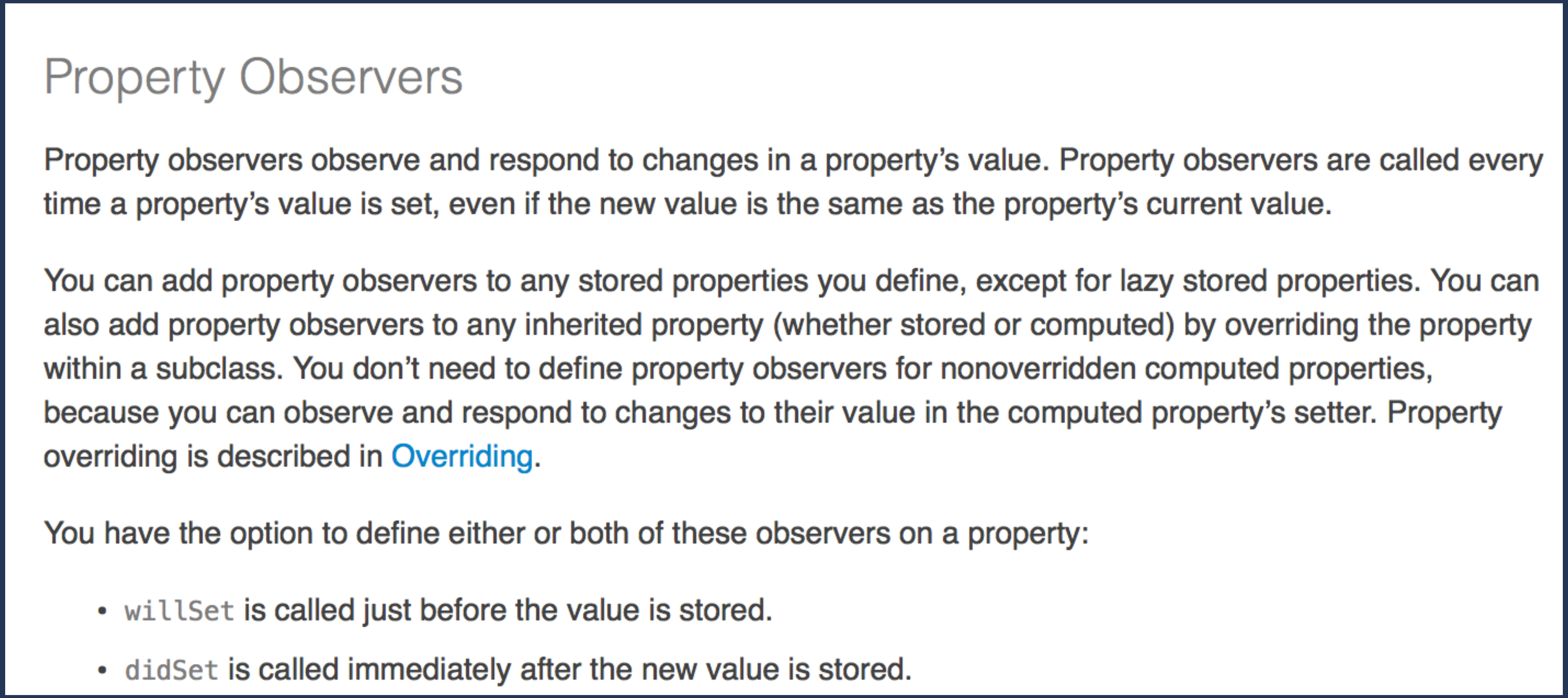
Applied Duality
Microsoft
NETFLIX
GitHub
SOUNDCLOUD
Trello
treehouse
SeatGeek
ooVoo
Couchbase
futurice
O.C.TANNER
airbnb

Rx is an API spec for "asynchronous programming with observable streams"

"Asynchronous programming with observable streams" 😧

Let's try breaking that down

- Async ✅
- Observable - `didSet` and `willSet` provide basic observable behaviour in Swift already (they're called property observers)



Property Observers

Property observers observe and respond to changes in a property’s value. Property observers are called every time a property’s value is set, even if the new value is the same as the property’s current value.

You can add property observers to any stored properties you define, except for lazy stored properties. You can also add property observers to any inherited property (whether stored or computed) by overriding the property within a subclass. You don’t need to define property observers for nonoverridden computed properties, because you can observe and respond to changes to their value in the computed property’s setter. Property overriding is described in Overriding.

You have the option to define either or both of these observers on a property:

- `willSet` is called just before the value is stored.
- `didSet` is called immediately after the new value is stored.

Advanced usage in lots of frameworks. Ember.js for example has lots of applications of the observer pattern.

```
Ember.Object.extend({
  valueObserver: Ember.observer('value', function(sender, key, value, rev) {
    // Executes whenever the "value" property changes
    // See the addObserver method for more information about the callback arguments
  })
});
```

```
Ember.Object.extend({
  valueObserver: Ember.observer('value', function(sender, key, value, rev) {
    // Executes whenever the "value" property changes
    // See the addObserver method for more information about the callback arguments
  })
});
```

- Observers are built on top of sequence types. Sort of behave like infinite lists.
- Flexible(you can send error events etc)
- Prevent callback hell
- Can be composed together

Not convinced?

Classic Rx example

- Let's make a search box with autocomplete
- Need to cancel a request when the text goes from ab -> abc
- Min 3 characters
- Exponential retry
- Wait before sending request, don't spam servers if user is typing quickly(say, 0.4 seconds)

- You would need 2 `Timer` objects in your `ViewController` - one for exponential retry, one for the 0.4 second timer
- Keep track of the current `URLSessionDataTask` so you can cancel it
- God knows how much state mutation
- Every time you add a feature, say a `UITableView` with the results, something would have to change in all this state.
- Eventually you will give up and rewrite and your code will still 💥

- The Rx way

```
searchTextField.rx.text
    .throttle(0.3, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query in
        API.getSearchResults(query)
            .retry(3)
            .startWith([]) // clears results on new search term
            .catchErrorJustReturn([])
    }
    .subscribe(onNext: { results in
      // bind to ui
    })
    .disposed(by: disposeBag)
```

- All local state
- Less code = best code 🎉

Are you not
Entertained?

Combine Network Requests

- Various parts of the UI(maybe even across controllers) need data from various requests
- Backend team won't design a composite API 😞
- Now, maintain a flag for every request, keep track of all requests, update parts of the UI. Messy state.
- GCD Service Groups solve the flag problem, but updating your UI is now closely coupled with updating your state.
- You could send `NSNotifications` with state updates, but that's not safe. *Anything* can trigger a notification post.
- Redo this for every feature. Rinse and repeat until heat death of the universe ☠️

The Rx Way

```
let userRequest: Observable<User> = API.getUser("me")
let friendsRequest: Observable<[Friend]> = API.getFriends("me")

Observable.zip(userRequest, friendsRequest) { user, friends in
    return (user, friends)
}
.observeOn(MainScheduler.instance)
.subscribe(onNext: { user, friends in
    // bind them to the user interface
})
.disposed(by: disposeBag)
```

- Note that if you need only one request for a part of your UI, you can update that separately 🎉 with a separate obeserver!
- Tomorrow, if you need to add a feature, just combine a few more signal observers!

Update multiple UI from single observer

- Use `Drivers`. Drivers are observables that only emit on the main thread and can't have error states.

```
let results = query.rx.text
    .throttle(0.3, scheduler: MainScheduler.instance)
    .flatMapLatest { query in
        fetchAutoCompleteItems(query)
            .observeOn(MainScheduler.instance)  // results are returned on MainScheduler
            .catchErrorJustReturn([])           // in the worst case, errors are handled
    }
    .shareReplay(1)                             // HTTP requests are shared and results replayed
                                                // to all UI elements
```

```
results
    .map { "\($0.count)" }
    .bind(to: resultCount.rx.text)
    .disposed(by: disposeBag)
results
    .bind(to: resultsTableView.rx.items(cellIdentifier: "Cell")) { (_, result, cell) in
        // bind data here
    }
    .disposed(by: disposeBag)
```

Sounds Cool, how do I make my own stuff?

Let's do a simple example with a URLRequest

```swift
extension Reactive where Base: URLSession {
    public func response(_ request: URLRequest) -> Observable<(Data, HTTPURLResponse)> {
        return Observable.create { observer in
            let task = self.dataTaskWithRequest(request) { (data, response, error) in
                guard let response = response, let data = data,
                    let httpResponse = response as? HTTPURLResponse else {
                    observer.on(.error(error ?? RxCocoaURLError.Unknown))
                    return
                }
                observer.on(.next(data, httpResponse))
                observer.on(.completed)
            }

            task.resume()
            //disposal code you shouldn't care about
        }
    }
}
```

- If you feel like experimenting with Rx, I'd suggest adding it slowly to an independent component of your app.
- Works 👌 with MVVM

Resources

- RxSwift Getting started
- Casey Liss' five part series
- Clean architecture in RxSwift
- Enemy of the state
- Rx Example
- Comparing RxSwift and ReactiveSwift

Clean Architecture

- Main points
- Divide App into 3 parts
  - Domain
  - Platform
  - Application

Domain

- Contains structs
- Also has "Use Cases" to do one specific things (protocol with abstract implementations)

```
swift
public protocol AllPostsUseCase {
func posts() -> Observable<[Post]>
}
```

Platform

- OS/Platform specific things(Core Data/Realm/Network)

```
final class RMPost: Object {
    dynamic var uid: String = ""
    dynamic var createDate: NSDate = NSDate()
    dynamic var updateDate: NSDate = NSDate()
    dynamic var title: String = ""
    dynamic var content: String = ""
}
```

- Concrete implementations of Use cases(save to DB)

Application

```
protocol ViewModelType {
    associatedtype Input
    associatedtype Output

    func transform(input: Input) -> Output
}
```

- View Controllers

```swift
final class PostsViewModel: ViewModelType {
    struct Input {
        let trigger: Driver<Void>
        let createPostTrigger: Driver<Void>
        let selection: Driver<IndexPath>
    }
    struct Output {
        let fetching: Driver<Bool>
        let posts: Driver<[Post]>
        let createPost: Driver<Void>
        let selectedPost: Driver<Post>
        let error: Driver<Error>
    }

    private let useCase: AllPostsUseCase
    private let navigator: PostsNavigator

    init(useCase: AllPostsUseCase, navigator: PostsNavigator) {
        self.useCase = useCase
        self.navigator = navigator
    }

    func transform(input: Input) -> Output {
    }
```

```
class PostsViewController: UIViewController {
    private let disposeBag = DisposeBag()

    var viewModel: PostsViewModel!
}
```

Shameless Plug

I'm working on OSS stuff right now

Interesting projects:
- Rewrite a reasonably complicated CLI tool in swift.
- Write a significant server side app in swift. A localization engine seems like an ideal choice.
- Write a parser of some sort (possibly, a query language -> Elastic search)
- Complicated stat stuff (Maybe a Support Vector Machine or a simple CNN that leverages Metal)

- Write a pure swift lib for the chromecast. It’s a major pain point that I want to get done with. Plus, I’ll finally get to work with protobufs(also good for overcast.fm)
- Build a watch auth service using APNS + bonjour. How will it work? Step 1 -> Send PNS with bonjour info. Step 2 -> notification on watch/phone. Step 3 -> send encrypted key thingy to mac using Bonjour.
- Do a ST <-> Atom <-> Xcode theme transpiler

Contact

- If you wanna help out with the projects, or just wanna say hi 👋
- @codeOfRobin on most places
- I haven't touched linkedIn in 3 years, please don't spam 🙈