/dev/world

Aug 29-31, 2016

# Advanced Patterns for Functional Reactive Programming in Swift


## Sebastian Grail

Canva

@sebastiangrail

github.com/sebastiangrail


AUC

In Partnership With

RMIT UNIVERSITY

# What we'll cover

- Chaining asynchronous tasks with `flatMap`
- Managing state with scan
- Handling errors
- Unfolding

# Chaining Network Requests

```
func doSomeNetworkStuff (arg: Foo, callback: (Bar) -> Void)

doSomeNetworkStuff(foo) { bar in
    doSomeOtherStuff(bar) { baz in
        whenDoesThisEverEnd(baz) { foobar in
            ohGodPleaseStop(foobar) { asdf in
                actuallyUseStuff(asdf)
            }
        }
    }
}
```

# Chaining Network Requests

```
func doSomeNetworkStuff (arg: Foo) -> SignalProducer<Bar, NoError>

doSomeNetworkStuff(foo)
    .flatMap(doSomeOtherStuff)
    .flatMap(thisIsntTooBad)
    .flatMap(iCouldGoOn)
    .startWithNext { result
        actuallyDoStuff(result)
    }
```

```
func createProducer <T, U>
                    (f: (T, U -> Void) -> Void)

}
```

```
func createProducer <T, U>
                    (f: (T, U -> Void) -> Void)
                    -> T -> SignalProducer<U, NoError> {




}
```

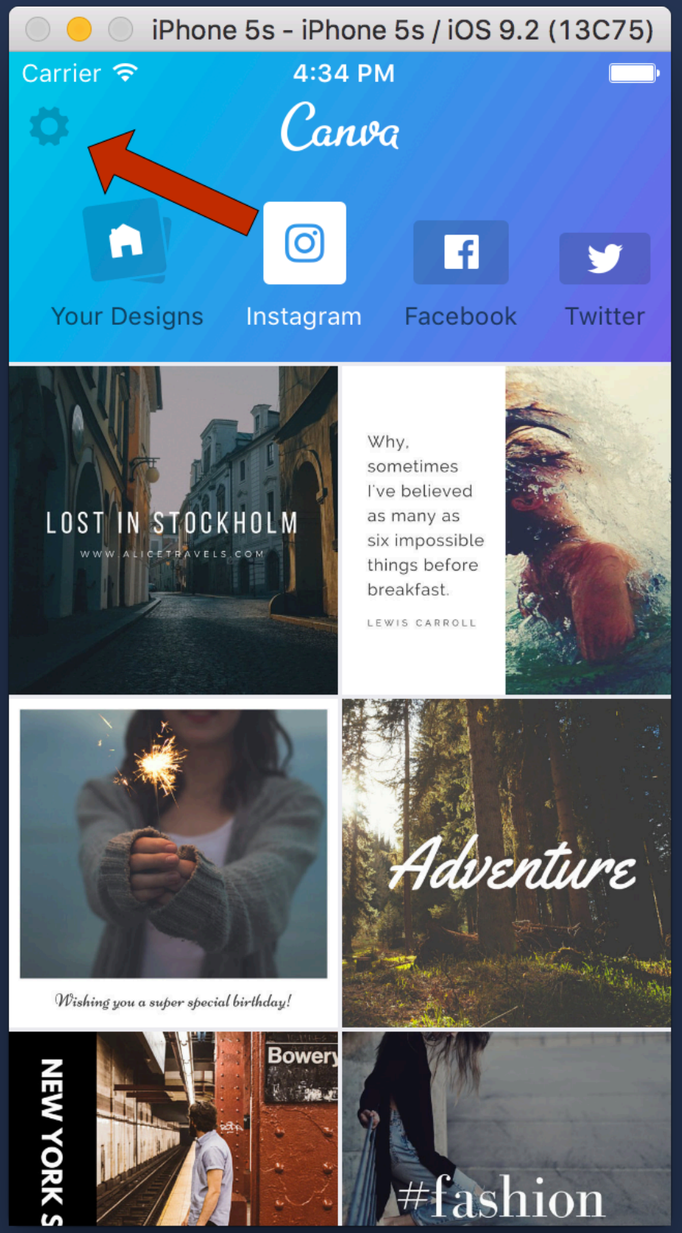
```
func createProducer <T, U>
                    (f: (T, U -> Void) -> Void)
                    -> T -> SignalProducer<U, NoError> {
    return { t in




    }
}
```

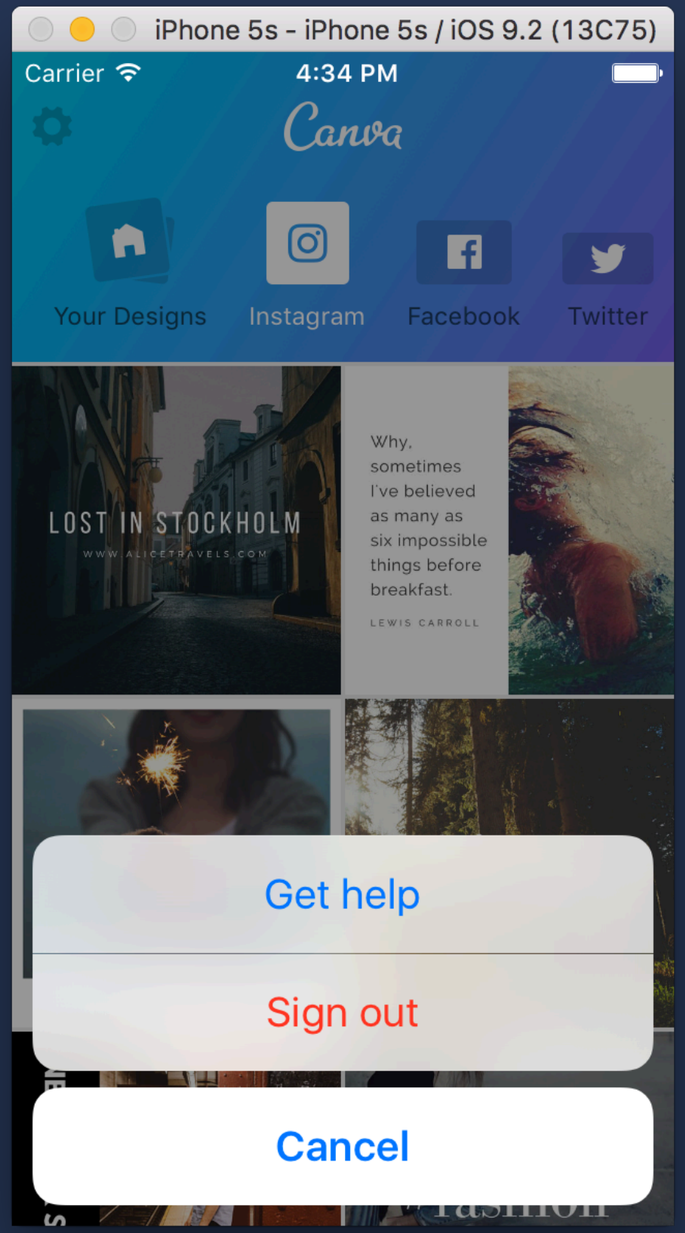
```
func createProducer <T, U>
                    (f: (T, U -> Void) -> Void)
                    -> T -> SignalProducer<U, NoError> {
    return { t in
        return SignalProducer { observer, _ in




        }
    }
}
```

```
func createProducer <T, U>
                    (f: (T, U -> Void) -> Void)
                    -> T -> SignalProducer<U, NoError> {
    return { t in
        return SignalProducer { observer, _ in
            f(t) { u in


            }
        }
    }
}
```

```
func createProducer <T, U>
                    (f: (T, U -> Void) -> Void)
                    -> T -> SignalProducer<U, NoError> {
    return { t in
        return SignalProducer { observer, _ in
            f(t) { u in
                observer.sendNext(u)
                observer.sendCompleted()
            }
        }
    }
}
```

# Observing events from a button before it is created


iPhone 5s - iPhone 5s / iOS 9.2 (13C75)
Carrier
4:34 PM
Canva
Your Designs
Instagram
Facebook
Twitter
LOST IN STOCKHOLM
WWW.ALICETRAVELS.COM
Why, sometimes I've believed as many as six impossible things before breakfast.
LEWIS CARROLL
Wishing you a super special birthday!
Adventure
NEW YORK
Bowery
#fashion


iPhone 5s - iPhone 5s / iOS 9.2 (13C75)
Carrier
4:34 PM
Canva
Your Designs
Instagram
Facebook
Twitter
LOST IN STOCKHOLM
WWW.ALICETRAVELS.COM
Why, sometimes I've believed as many as six impossible things before breakfast.
LEWIS CARROLL
Get help
Sign out
Cancel

```
SignalProducer<(), NoError> { (observer, _) in
    let alert = UIAlertController(... preferredStyle: .ActionSheet)

    rootVC.presentViewController(alert, animated: true, completion: nil)
}
```

```
SignalProducer<(), NoError> { (observer, _) in
    let alert = UIAlertController(... preferredStyle: .ActionSheet)

    alert.addAction(UIAlertAction(title: "Sign out", style: .Destructive) { _ in


    })

    rootVC.presentViewController(alert, animated: true, completion: nil)
}
```

```
SignalProducer<(), NoError> { (observer, _) in
    let alert = UIAlertController(... preferredStyle: .ActionSheet)

    alert.addAction(UIAlertAction(title: "Sign out", style: .Destructive) { _ in
        observer.sendNext()
        observer.sendCompleted()
    })

    rootVC.presentViewController(alert, animated: true, completion: nil)
}
```

```
SignalProducer<(), NoError> { (observer, _) in
    let alert = UIAlertController(... preferredStyle: .ActionSheet)

    alert.addAction(UIAlertAction(title: "Sign out", style: .Destructive) { _ in
        observer.sendNext()
        observer.sendCompleted()
    })

    alert.addAction(UIAlertAction(title: "Cancel", style: .Cancel) { _ in
        observer.sendCompleted()
    })

    rootVC.presentViewController(alert, animated: true, completion: nil)
}
```

```
let logoutSignal = cogButton.flatMap(.Latest) { _ in
    return SignalProducer<(), NoError> { (observer, _) in
        let alert = UIAlertController(... preferredStyle: .ActionSheet)

        alert.addAction(UIAlertAction(title: "Sign out", style: .Destructive) { _ in
            observer.sendNext()
            observer.sendCompleted()
        })

        alert.addAction(UIAlertAction(title: "Cancel", style: .Cancel) { _ in
            observer.sendCompleted()
        })

        rootVC.presentViewController(alert, animated: true, completion: nil)
    }
}
```

# Filtering and mapping with `flatMap`

```
let someSignal: SignalProducer<Int, Error> = ...

someSignal.flatMap { n in
    if n % 2 == 0 {
        return SignalProducer(value: n*n)
    } else {
        return SignalProducer.empty
    }
}
```

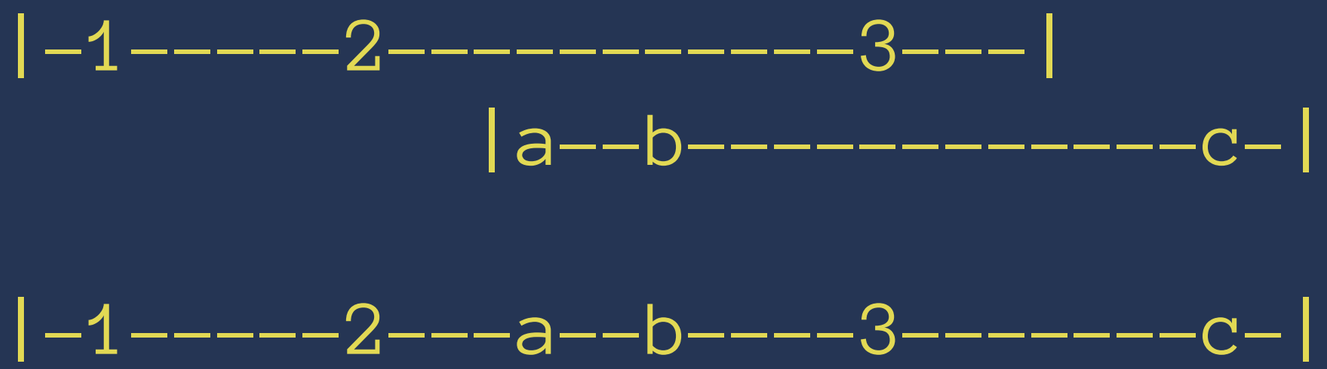
- On button tap
- Start network request
- If payemnt required
  - Present UI with confirm/cancel
  - On confirm start new request
  - On cancel end the whole chain

```
button
    .flatMap(exportDocument)
    .flatMapError { error in
        guard error == .PaymentRequired else {
            return SignalProducer(error: error)
        }
        return showPaymentUI()
            .flatMap(processPayment)
            .then(exportDocument)
    }
```

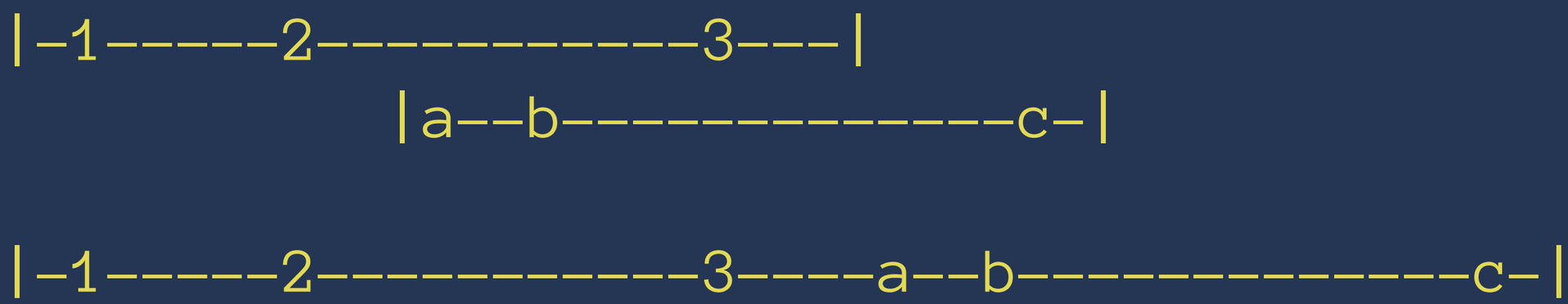
# Understanding flatten strategies

- `.Merge`: All values from all producers are forwarded immediately
- `.Concat`: Values are sent in the order of the producers
- `.Latest`: Only values from the latest producer are forwarded

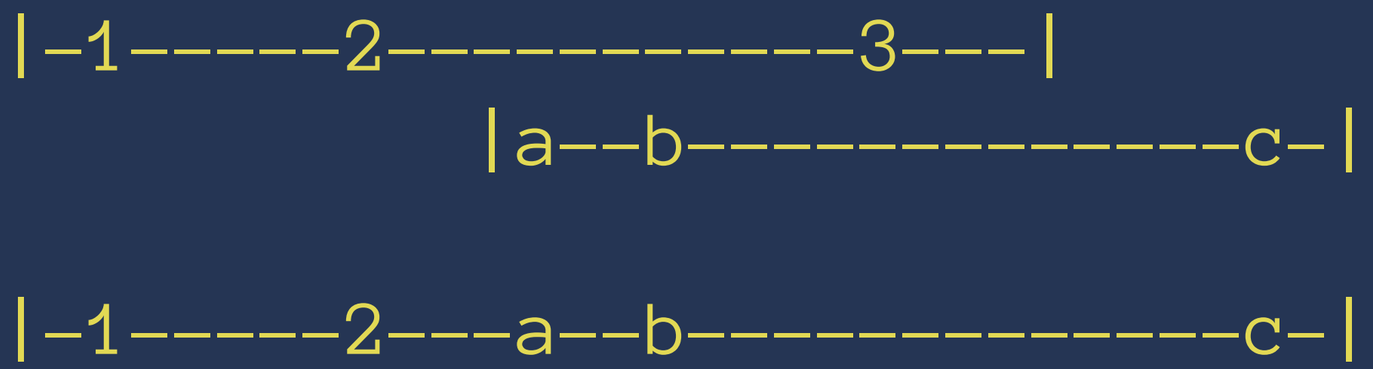
# .Merge



# .Concat



# .Latest



# Counting button taps

- button with target/action
- counter variable
- method for the button target that mutates the couter

# Counting button taps

```
var counter = 0
let counterSignal = plusButton.map { _ -> Int in
    counter += 1
    return counter
}
```

# Counting button taps

```
var counter = 0
let counterSignal = plusButton.map { _ -> Int in
    counter += 1
    return counter
}
```

- Observation changes the signal
- Different observers get different streams of values

# Summing an array of `Int`s

```
[1,2,3]
```

# Summing an array of `Int`s

```
[1,2,3].reduce(0, +) // => 6
// (((0 + 1) + 2 ) + 3)
```

# Summing an array of `Int`s with intermediate values

```
[1,2,3].scan(0, +) // => [0,1,3,6]
// (((0 + 1) + 2 ) + 3)
```

# Keeping state between events with `scan`

```
// Array
func reduce <U>
            (initial: U, combine: (U, Value) -> U)
            -> U
```

# Keeping state between events with `scan`

```
// Array
func scan    <U>
             (initial: U, combine: (U, Value) -> U)
             -> [U]
```

# Keeping state between events with `scan`

```
// Array
func scan <U>
          (initial: U, combine: (U, Value) -> U)
          -> [U]

// SignalProducer
func scan <U>
          (initial: U, combine: (U, Value) -> U)
          -> SignalProducer<U, Error>
```

# Keeping state between events with `scan`

```
plusButton
    .scan(0) { runningTotal, _ in runningTotal + 1 }
    .startWithNext { (x) in
        print("total number of taps: \(x)")
    }
```

# Keeping state between events with `scan`

```
plusButton.map { 1 }
    .scan(0, +)
    .startWithNext { (x) in
        print("total number of taps: \(x)")
    }
```

# Keeping state between events with `scan`

```
SignalProducer.merge([plusButton.map  {  1 },
                      minusButton.map { -1 }])
    .scan(0, +)
    .startWithNext { (x) in
        print(x)
    }
```

# Switch between live and snapshot rendering

```
renderingMode.producer.scan((nil, .Live)) { old, newMode in
    switch (old.mode, newMode) {
        case (.Live, .Snapshot):
            return (self.contentView.snapshotViewAfterScreenUpdates(false), .Snapshot)
        default:
            return (old.snapShotView, newMode)
    }
}.startWithNext(
    switch $0 {
    case (.Some(let oldView), .Live):
        oldView.removeFromSuperview()
    case (.Some(let newView), .Snapshot) where newView.superview == nil:
        self.contentView.addSubview(newView)
    }
})
```

# Handling Errors

- Once an error occurs, a signal (and all its derived signals) stop sending values
- `signal.flatMap(.Latest, f)` fails when the producer returned by `f` fails
- Completes when both `signal` and the latest result of `f` complete

# Handling Errors

```
login: (String, String) -> SignalProducer<User, LoginError>

name.combineLatestWith(password)
    .sampleOn(button)
    .flatMap(.Latest) { x,y in
        login(x,password: y)
    }
```

# Handling Errors with `flatMapError`

```
func flatMapError <F>
                  (handler: Error -> SignalProducer<Value, F>)
                  -> SignalProducer<Value, F>
```

- Starts a new producer when an error occurs
- Changes the type of the error
- Keeps the type of the value

# Handling Errors with `materialize`

```
func materialize()
     -> SignalProducer<Event<Value, Error>, NoError>
```

- Moves all events into the value
- Errors on the original signal complete the resulting signal

# Handling Errors with `materialize`

```
name.combineLatestWith(password)
    .sampleOn(button)
    .flatMap(.Latest) { x,y in
        login(x,password: y)
    }
```

# Handling Errors with `materialize`

```
name.combineLatestWith(password)
    .sampleOn(button)
    .flatMap(.Latest) { x,y in
        login(x,password: y).materialize()
    }
```

# Handling Errors with `materialize`

```
name.combineLatestWith(password)
    .sampleOn(button)
    .flatMap(.Latest) { x,y in
        login(x,password: y).materialize()
    }
```

```
SignalProducer<User, LoginError>
SignalProducer<Event<User, LoginError>, NoError>
```

# Handling Errors with `retry`

```
signalThatCanFail
    .retry(n)
```

- Swallows up to n errors
- Fails after nth error
- Errors need to be handled before `retry`

# Unfolding

- `reduce` is also known as `fold`
- `fold` goes from a series of values to one value
- `unfold` goes from one value to a series of values

# Unfolding

```
func getDesigns (token: Token)
                -> SignalProducer<([Design], Token), Error>
```

- each call returns an array of designs and a continuation token
- token from one call loads the next set of designs

# Unfolding

```
func unfold <T, V, E: ErrorType>
     (f: T -> SignalProducer<(V, T), E>)
```

# Unfolding

```
func unfold <T, V, E: ErrorType>
     (f: T -> SignalProducer<(V, T), E>, initial: T)
```

# Unfolding

```
func unfold <T, V, E: ErrorType>
     (f: T -> SignalProducer<(V, T), E>, initial: T)
     -> SignalProducer<V, E>
```

# Unfolding

```
func unfold <T, V, E: ErrorType>
     (f: T -> SignalProducer<(V, T), E>, initial: T)
     -> SignalProducer<V, E>
{
    return f(initial)
}
```

# Unfolding

```
func unfold <T, V, E: ErrorType>
     (f: T -> SignalProducer<(V, T), E>, initial: T)
     -> SignalProducer<V, E>
{
    return f(initial).flatMap(.Concat) {
    }
}
```

# Unfolding

```
func unfold <T, V, E: ErrorType>
     (f: T -> SignalProducer<(V, T), E>, initial: T)
     -> SignalProducer<V, E>
{
    return f(initial).flatMap(.Concat) {
        return unfold(f, initial: $1).startWith($0)
    }
}
```


Thanks!