Thinking in RxSwift way

Krunoslav Zaher

Arrows

Just in case …

```
let results = searchText
      .map { $0.trimmingCharacters(in: .whitespaces) }
      .throttle(0.3, scheduler: MainScheduler.instance)
      .distinctUntilChanged()
      .flatMapLatest { query -> Observable<[Repository]> in
          if query.isEmpty {
              return Observable.just([])
          } else {
              return api(query)
                  .catchErrorJustReturn([])
                  .observeOn(MainScheduler.instance)
          }
      }
```

Why? I’m lazy …

I enjoy deleting code …

Pure functions?

What about Reactive systems

- taps on screen
- network call results
- timers
- async operations
- …

Your systems are already reactive

What do all of them have in common?

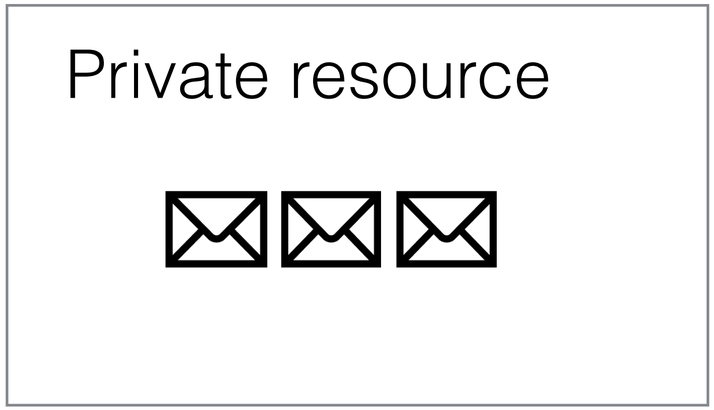
It’s just a callback

```
// Observable sequence
public protocol ObservableType {
    associatedtype Message
    func subscribe(observer: (Message) -> ()) -> Disposable
}

// Observable sequence
public protocol ObservableType {
    associatedtype Element
    func subscribe(observer: (Event<Element>) -> ()) -> Disposable
}

public enum Event<Element> {
    case next(Element)
    case error(Swift.Error)
    case completed
}

typealias Disposable = () -> ()
```

How Rx works

```
// Observable sequence
public protocol ObservableType {
    associatedtype Message
    func subscribe(observer: (Message) -> ()) -> Disposable
}
```

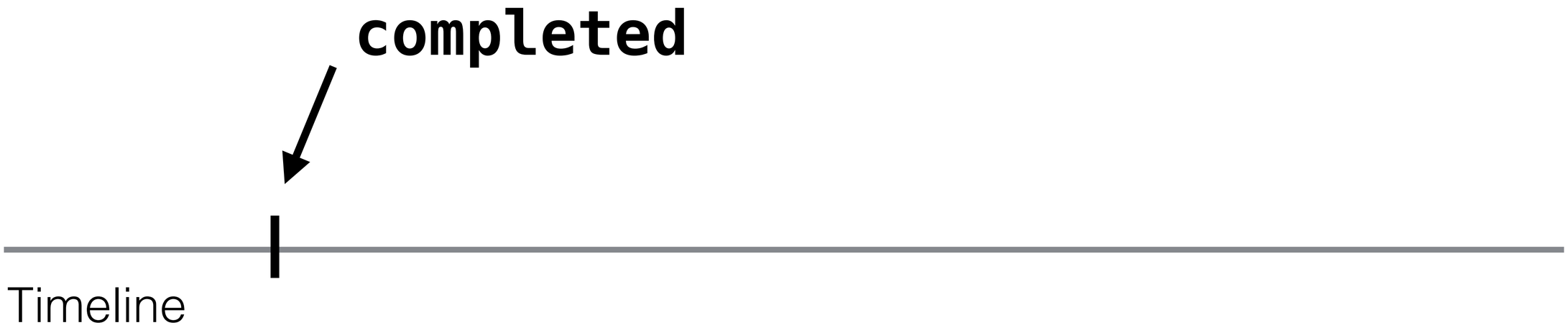
```
// Callback
(Message) -> ()
```




```
// Subscription (Disposable type)
() -> ()
```

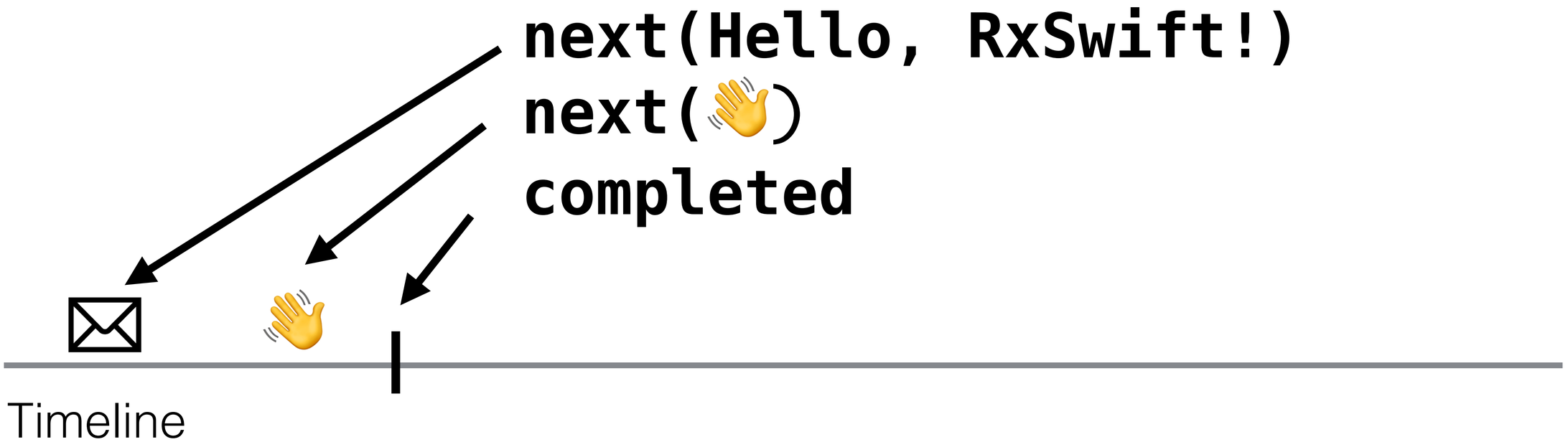
Timeline

```
Observable.empty()
      .subscribe {
            print($0)
        }
```



```
Observable.just("Hello, RxSwift!")
      .subscribe {
            print($0)
        }
```

```
Observable.of("Hello, RxSwift!", "👋")
    .subscribe {
        print($0)
    }
```



```
Observable<Int>.interval(1.0, scheduler: MainScheduler.instance)
    .subscribe {
        print($0)
    }
```

next(0)
next(1)
next(2)
next(3)
next(4)
next(5)
next(6)
…

```
let letters = Observable<Int>.interval(1.0, scheduler: scheduler)
    .map { String(format: "%c", 98 + $0) }
    .startWith("a")

let numbers = Observable<Int>.interval(2.0, scheduler: scheduler)
    .map { $0 + 1 }
    .startWith(0)

Observable.combineLatest(letters, numbers)
    .subscribe {
        print($0)
    }
```

```
next(("a", 0))
next(("b", 0))
next(("b", 1))
next(("c", 1))
next(("c", 2))
next(("d", 2))
…
```

Closure

```
let results = searchText
      .map { $0.trimmingCharacters(in: .whitespaces) }
      .throttle(0.3, scheduler: MainScheduler.instance)
      .distinctUntilChanged()
      .flatMapLatest { query -> Observable<[Repository]> in
          if query.isEmpty {
              return Observable.just([])
          } else {
              return api(query)
                  .catchErrorJustReturn([])
                  .observeOn(MainScheduler.instance)
          }
      }
```

```
searchText
      .map { $0.trimmingCharacters(in: .whitespaces) }
      .throttle(0.3, scheduler: MainScheduler.instance)
      .distinctUntilChanged()
      .flatMapLatest { query -> Observable<[Repository]> in
          if query.isEmpty {
              return Observable.just([])
          } else {
              return api(query)
                  .catchErrorJustReturn([])
                  .observeOn(MainScheduler.instance)
          }
      }
      .subscribe(onNext: { self.latestValue = $0 })
```

```
firstSequence
    .subscribe(onNext: { first in
        secondSequence(param: first)
            .subscribe(onNext: { result in
                print(result)
            })
    })
```

```
firstSequence
    .flatMapLatest { first in
        return secondSequence(param: first)
    }
```

How does Rx help?

We can take all of the ugly parts out of your systems

- mutations
- locking
- concurrency
- control state machines
- error propagation
- …

… and leave only pure functions in your code base

Steep learning curve

The interface itself is pretty simple …
but how many operators are there exactly?

Let’s recap

We want a composable way to represent reactive systems
-> Rx has awesome compositional properties

Reactive systems are based on callbacks
-> if you have callback you can wrap it with Rx

Rx is just a callback
Rx <-> callbacks

I could talk all day

Access control by using lambdas and Rx

Simple visualization and reasoning

Removing transient state machines from your code

Sharing side effects

Compile time proof that system is always ready to receive events

Resource management with Rx

…

Let’s build reactive systems

What is the semantics of observable sequences?
What do observable sequence elements represent? Events? State?

How to Introduce Rx partially? (legacy apps)

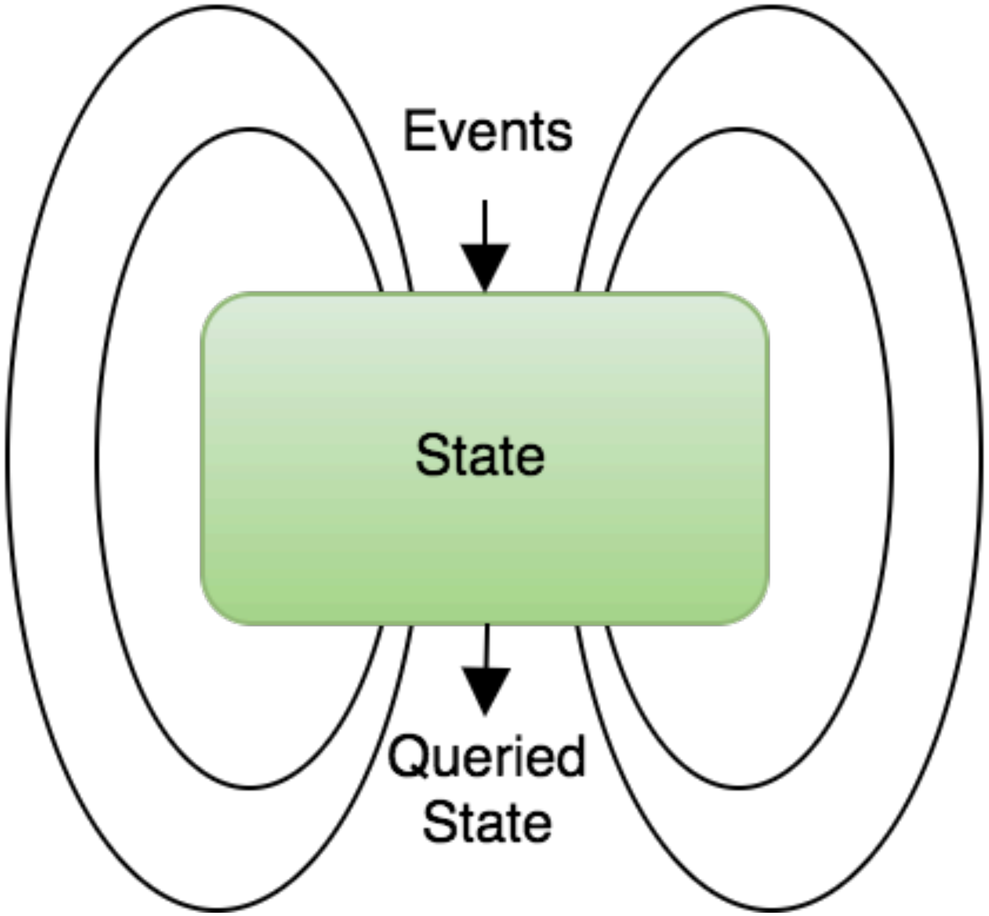
Architectures with Rx? (managing state)

How to introduce Rx to new team members?

How to simplify code reviews and make code idiomatic?

How to normalize Rx stateful solutions?

Feedback loops (RxFeedback)



Rx modeled feedback loops performing
Effects, IO & Resource management

- UI
- Networking
- Timers
- Bluetooth
- ...

https://github.com/kzaher/RxFeedback

Arrows

Scan operator

CQRS (Command query responsibility segregation)

Elm

Unidirectional architecture

Cross Platform code

ReactorKit

Databases in general

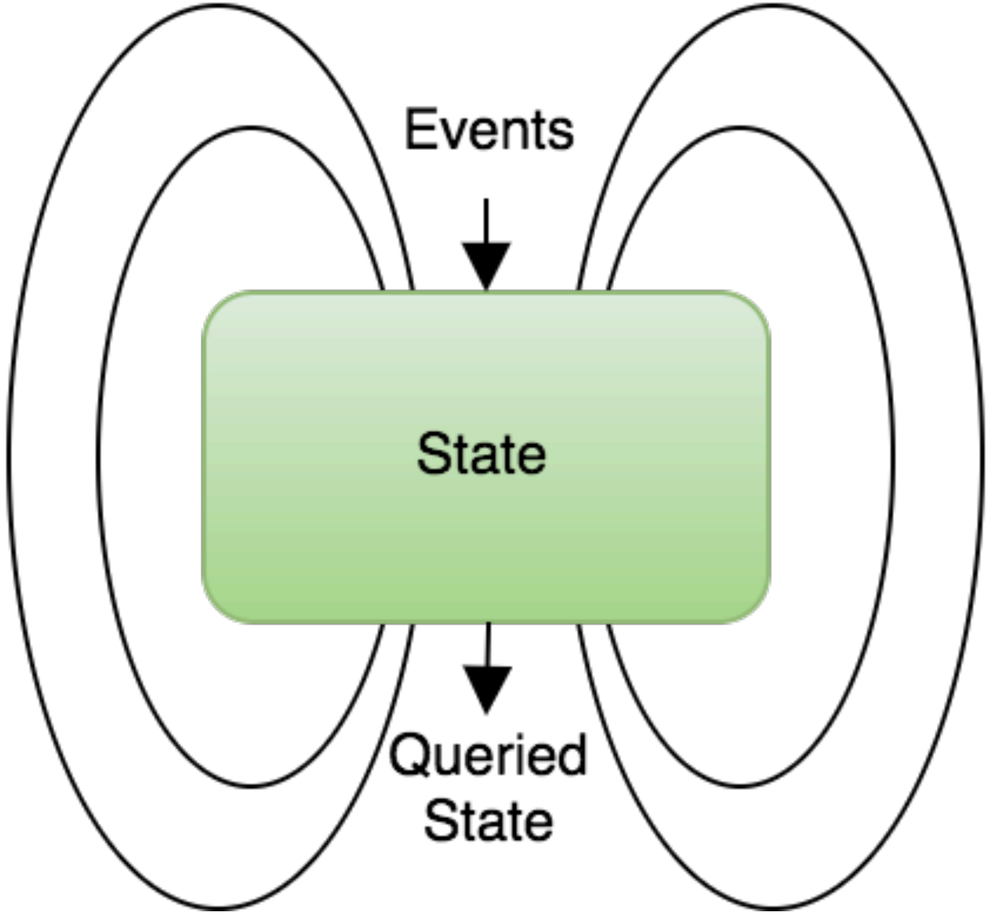
Event Sourcing

Redux Observable

Apache Kafka

Redux

RxFeedback



State transition

```
func system<State, Event>(
        initialState: State,
        reduce: @escaping (State, Event) -> State,
        feedback: (Observable<State>) -> Observable<Event>...
) -> Observable<State>
```

Query

Commands

```
let replaySubject = ReplaySubject<State>.create(bufferSize: 1)

            let events: Observable<Event> = Observable
               .merge(feedback.map { $0(replaySubject.asObservable()) })

            return events.scan(initialState, accumulator: reduce)
                .startWith(initialState)
                .do(onNext: { output in
                    replaySubject.onNext(output)
                })
```

Demo

Community

- RxSwift (https://github.com/ReactiveX/RxSwift)
  - concept that enables composition
- RxSwiftCommunity (https://github.com/RxSwiftCommunity)
  - reusable components and extensions
- RxFeedback (https://github.com/kzaher/RxFeedback)
  - architecture
  - use cases for that architecture (Paging, Bluetooth, OAuth …)

Next Steps

- Porting `RxFeedback` to Kotlin
- Porting `Driver`s to Kotlin
- Porting `RxFeedback` to RxJS
- Examples with RxFeedback and React
- Open sourcing observable key value store and other examples
- …