Functional Reactive Programming
using RxSwift

Sylvain Rebaud
http://github.com/c0diq

FRP Principles

RxSwift Building Blocks

Example Walkthrough

Final Thoughts

Functional Reactive Programming

Functional Programming

Immutable

Stateless

Predictable

Testable

Reactive Programming

What instead of *How*

Derived state

Data flow

Why FRP?

“UIs are big, messy, mutable, stateful bags of sadness.”

–Josh Abernathy

Every line of code we write is executed in reaction to an event…

… but these events come in many different forms

How?

Functional Reactive Programming provide a common interface for all events

… this allows us to define a language for manipulating, transforming and coordinating events

Rx Building Blocks

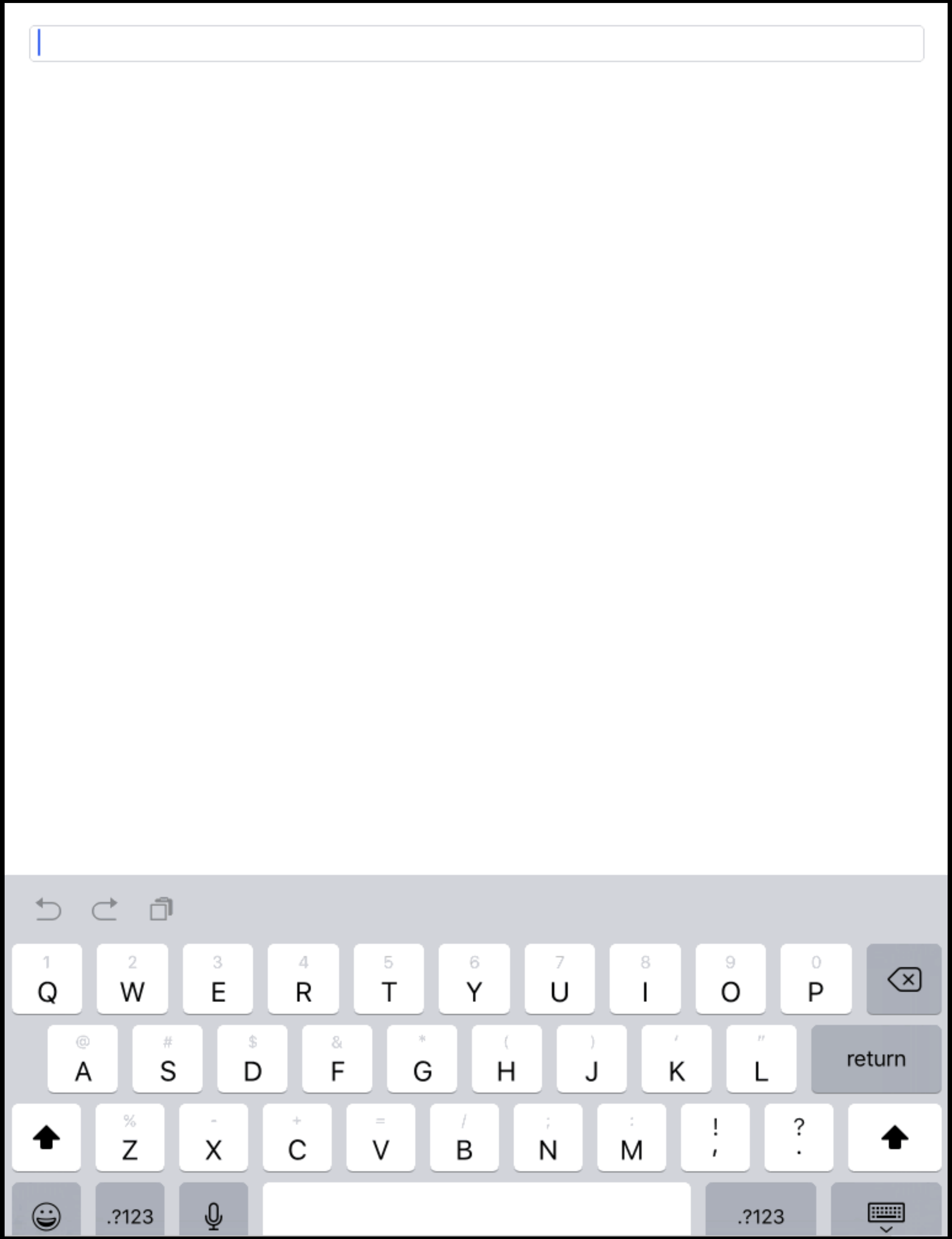
Observables

Operators

Schedulers

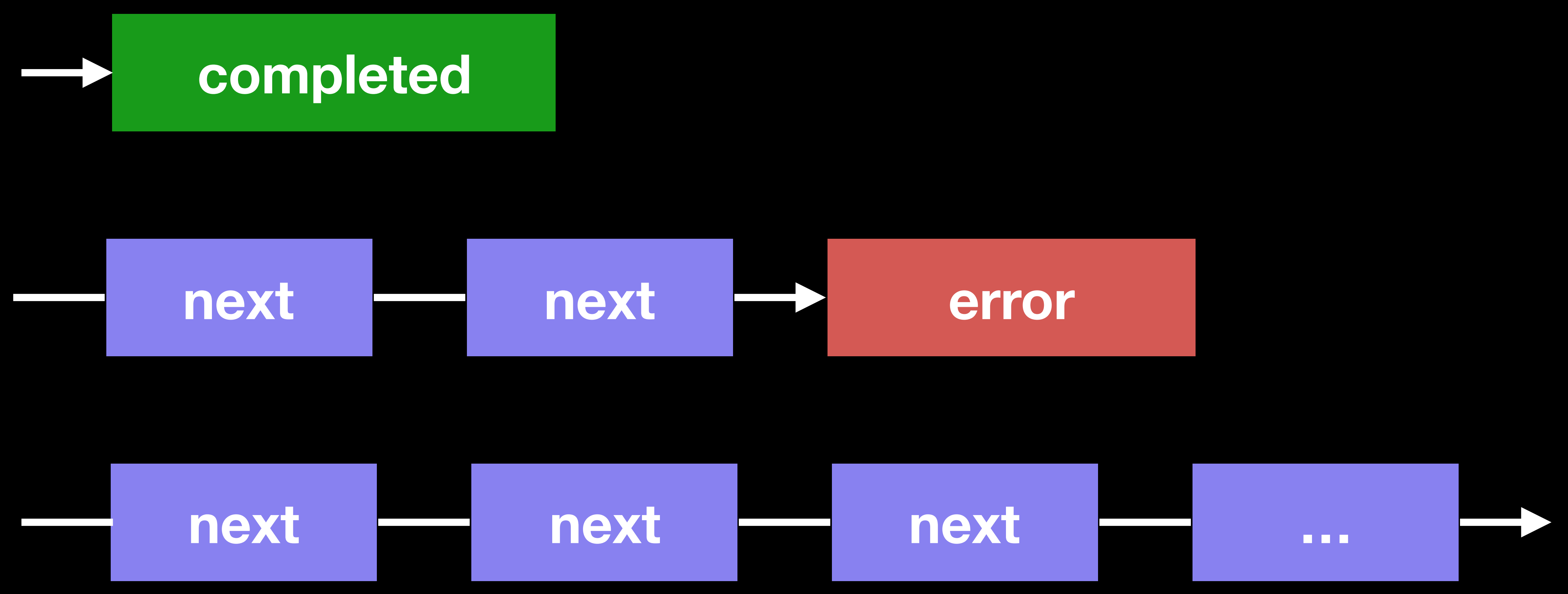
Observables

```swift
let text = textField.rx.text

text.subscribe(onNext: {
    print("\($0 ?? "")")
})
```


1 2 3 4 5 6 7 8 9 0
Q W E R T Y U I O P
@ # $ & * ( ) ' "
A S D F G H J K L
return
% - + = / ; :
Z X C V B N M ! , ? .
.?123
.?123

Observables emit a sequence of events to their observers

… they emit none, one or more next events, optionally followed by an error or completed



Observable Everything!

What are events?
What do they look like?

Anything!

Text from a UITextField control

JSON data from a Network Response

Notification

Number

String

Button Tap

…

```swift
/// Represents a sequence event.
///
/// Sequence grammar:
/// **next\* (error | completed)**
public enum Event<Element> {
    /// Next element is produced.
    case next(Element)

    /// Sequence terminated with an error.
    case error(Swift.Error)

    /// Sequence completed successfully.
    case completed
}
```

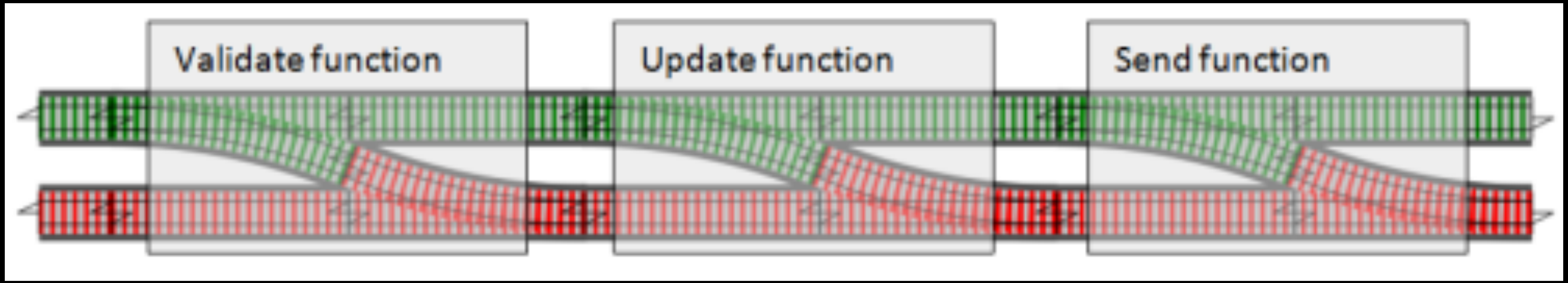
What about errors?

Errors

Railway Oriented Programming

Errors are passed on to the next step in the stream

If any point of the stream fails, we can handle this in one place.

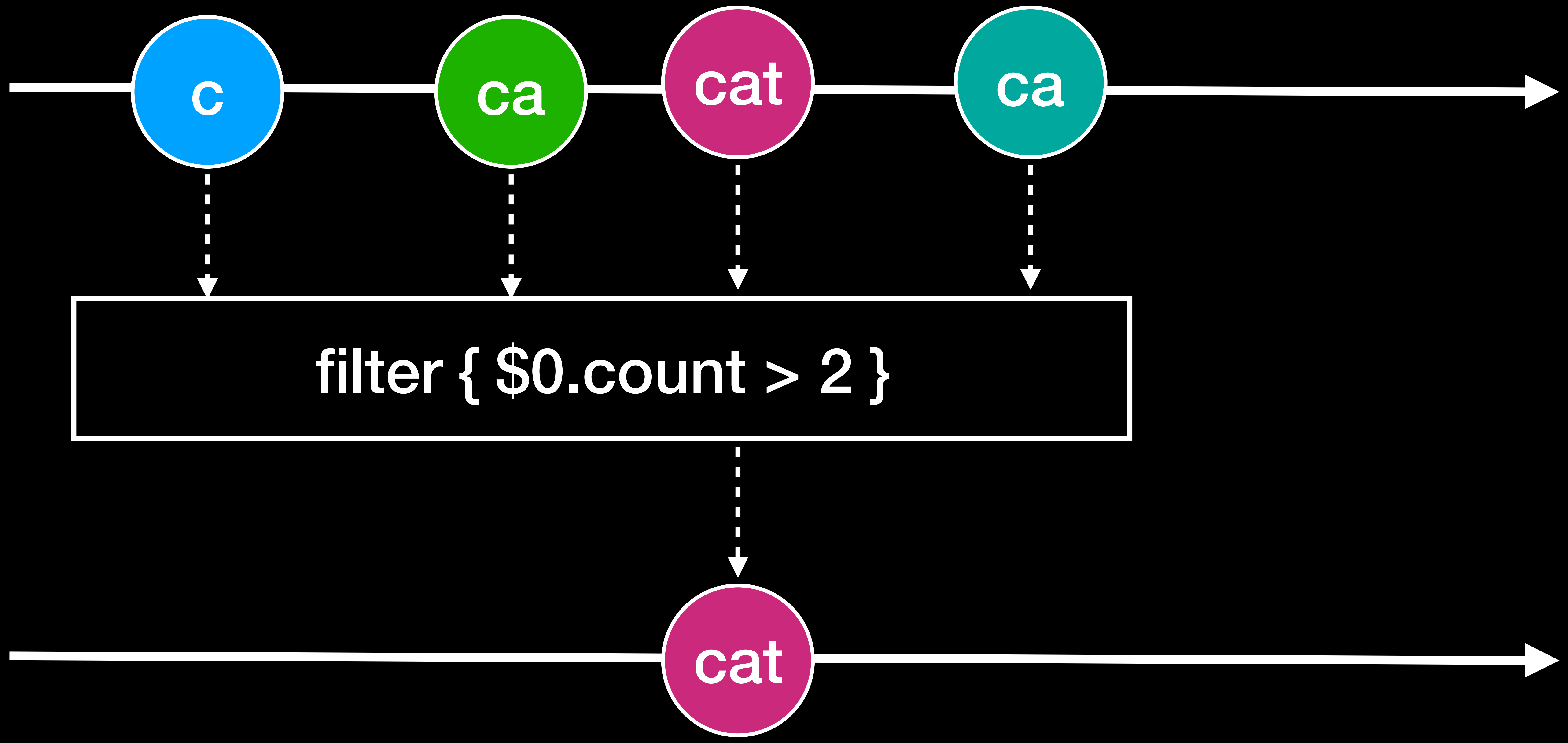


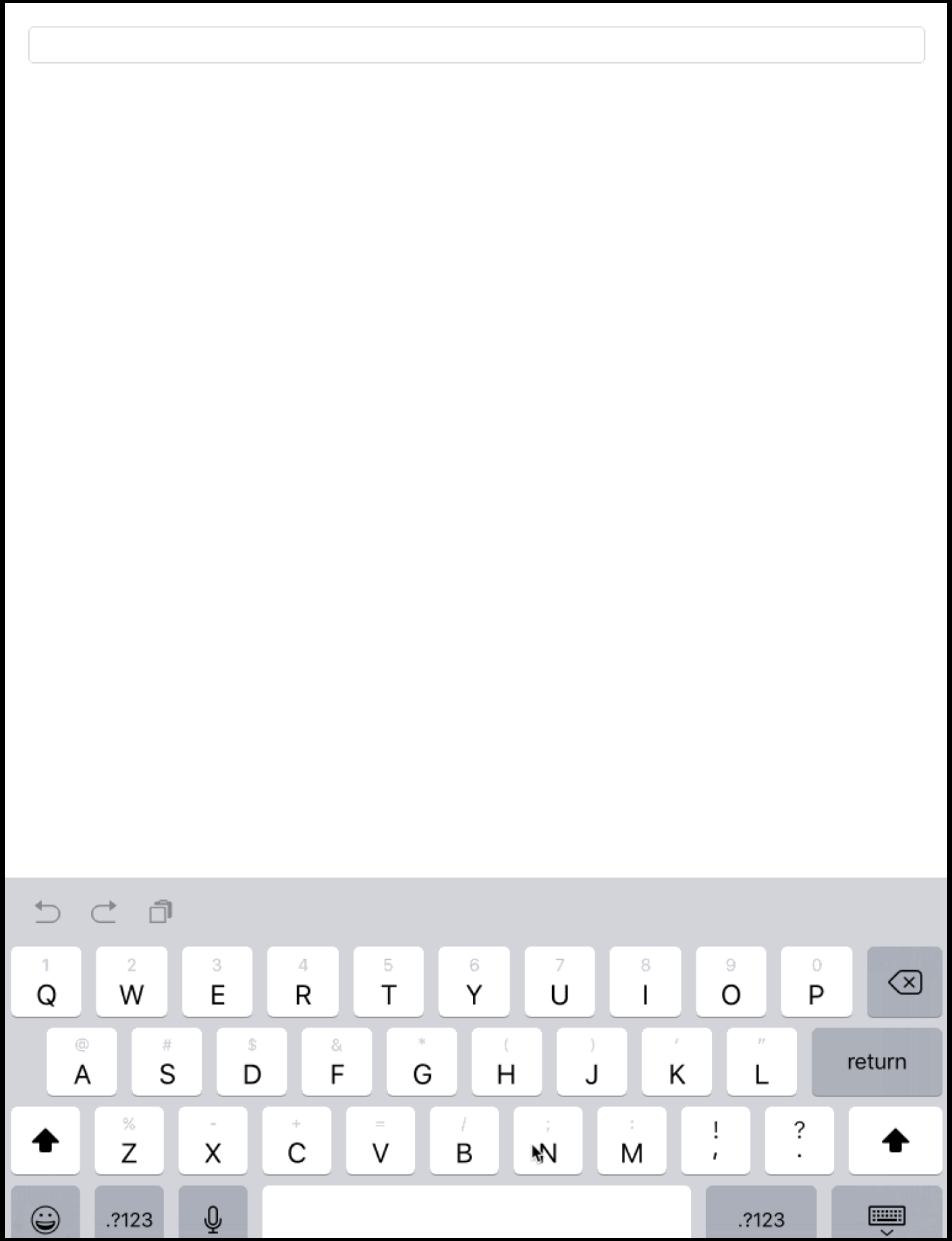
Operators

Operators

- delay
- skip
- take
- takeUntil
- flatMap
- concat
- merge
- startWith
- debounce
- reduce
- ignoreElements
- interval
- timeout
- buffer
- retry
- zip
- combineLatest
- never
- switchLatest
- distinctUntilChanged
- empty
- just
- error
- throttle
- of
- from
- sample
- scan
- window
- delaySubscription
- map
- filter
- flatMapLatest
- do
- withLatestFrom
- …

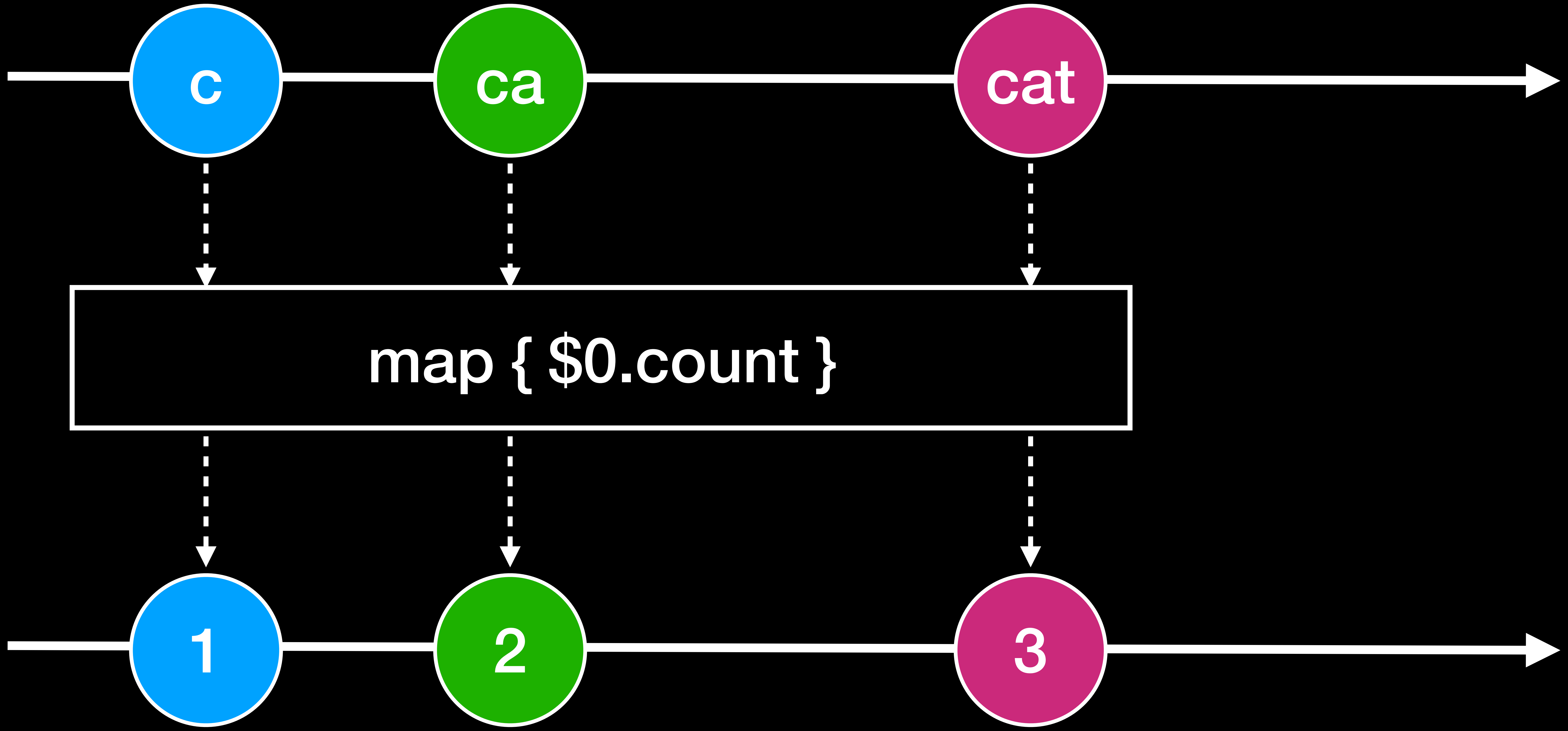
```swift
let text = textField.rx.text.orEmpty

let filteredText = text.filter {
    $0.count > 3
}

filteredText.subscribe(onNext: {
    print("\($0)")
})
```

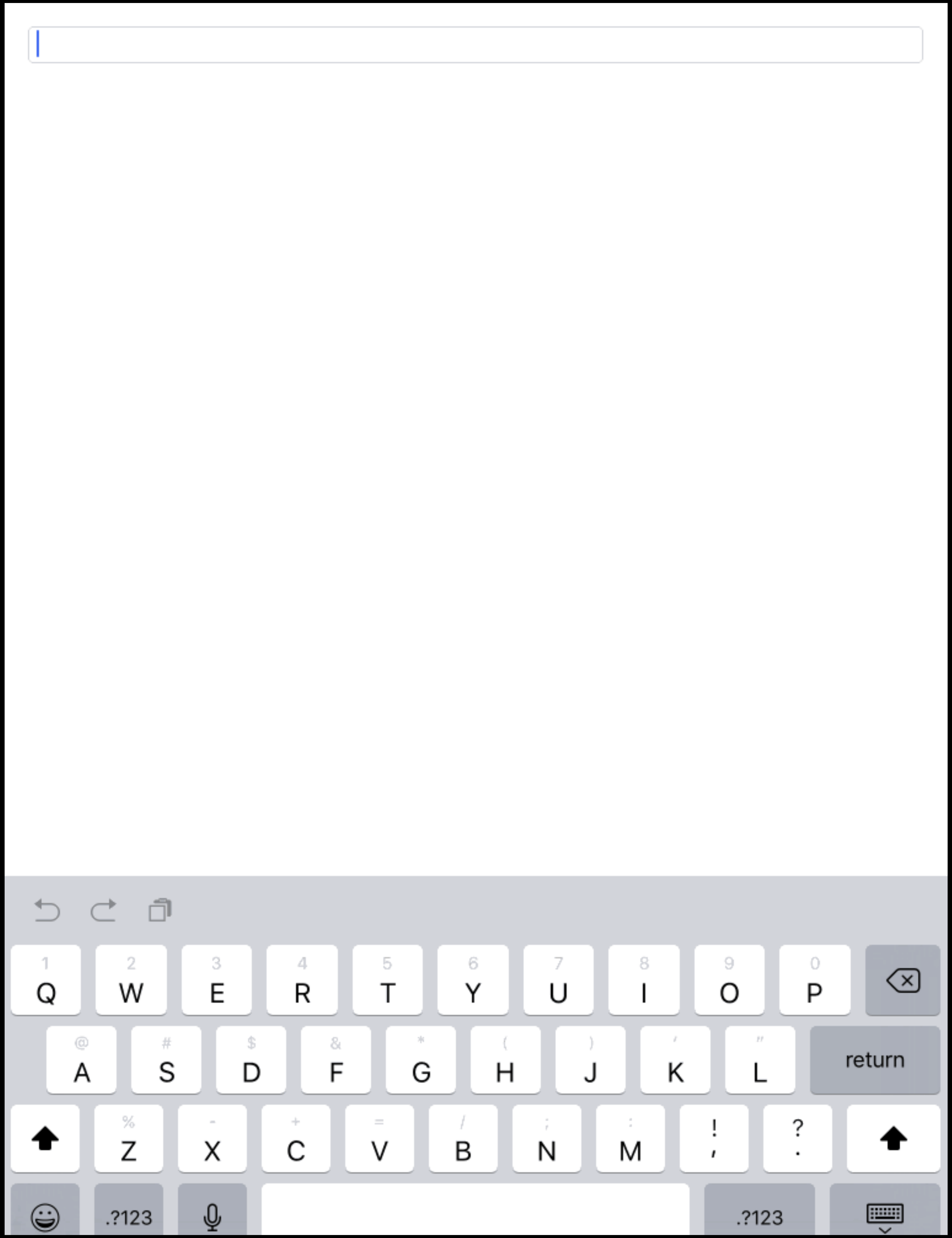

c
ca
cat
ca
filter { $0.count > 2 }
cat



1 2 3 4 5 6 7 8 9 0
Q W E R T Y U I O P
@ # $ & * ( ) ' "
A S D F G H J K L
return
% - + = / ; :
Z X C V B N M ! , ? .
.?123
.?123

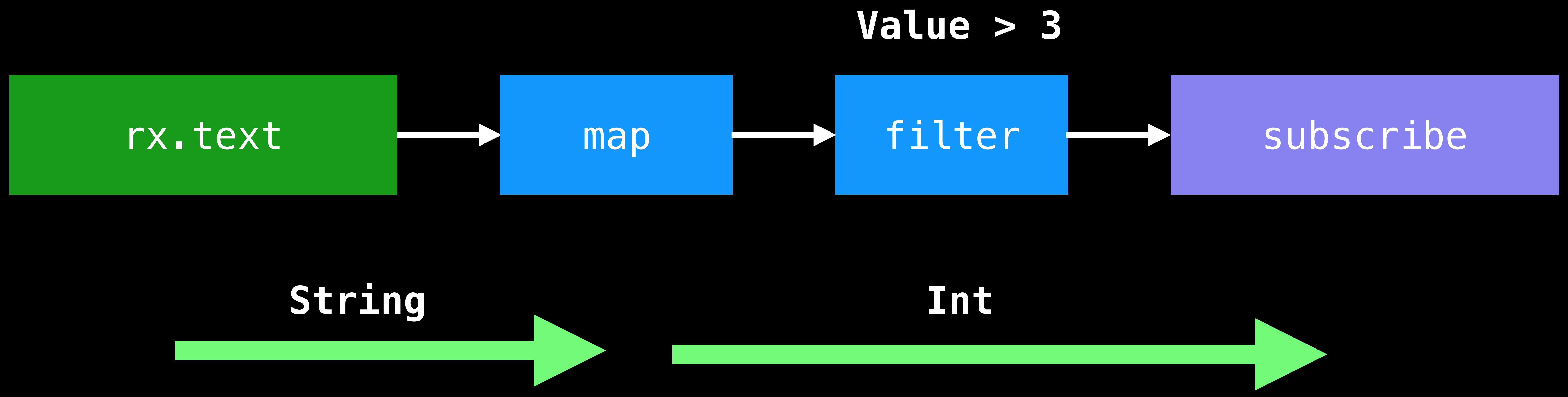
```swift
let text = textField.rx.text.orEmpty

let textLength = text.map {
    $0.count
}

textLength.subscribe(onNext: {
    print("\($0)")
})
```


c
ca
cat
map { $0.count }
1
2
3


1 2 3 4 5 6 7 8 9 0
Q W E R T Y U I O P
@ # $ & * ( ) ' "
A S D F G H J K L
return
% - + = / ; :
Z X C V B N M ! , ? .
.?123
.?123

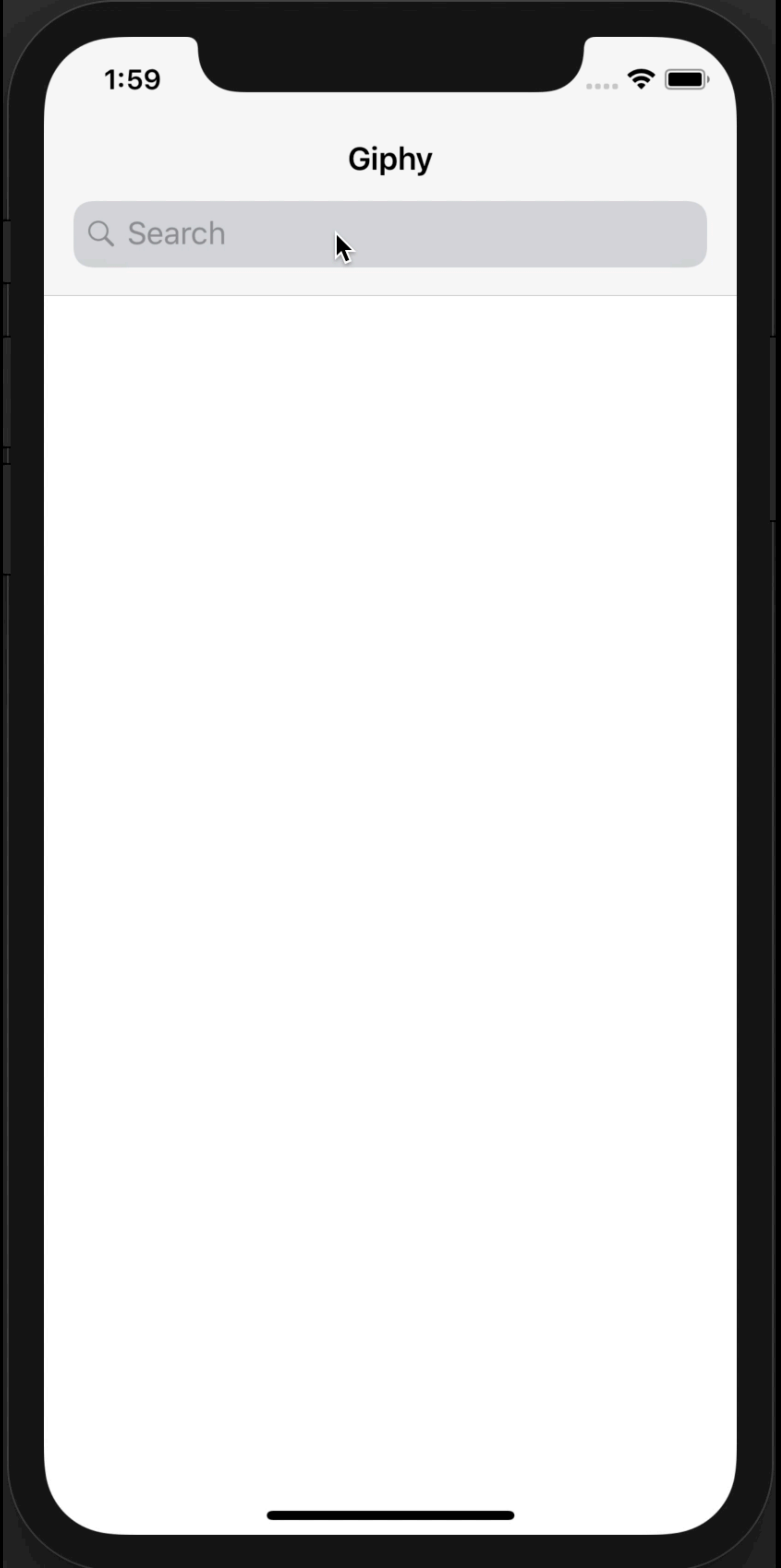
```swift
let text = textField.rx.text.orEmpty

let textLength = text.map {
    $0.count
}

let filteredLength = textLength.filter {
    $0 > 3
}

filteredLength.subscribe(onNext: {
    print("\($0)")
})
```

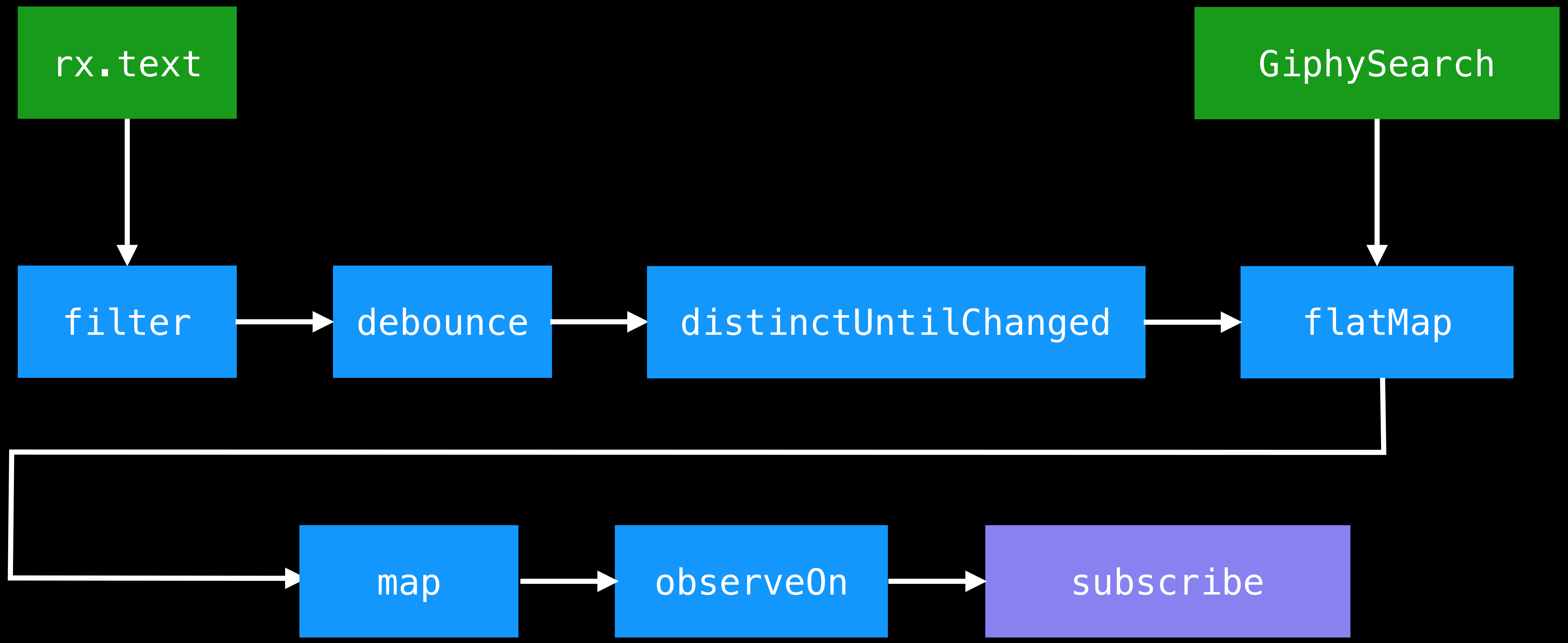
```swift
textField.rx.text.orEmpty
    .map { $0.count }
    .filter { $0 > 3 }
    .subscribe(onNext: {
        print("\($0)")
    })
```


Value > 3
rx.text
map
filter
subscribe
String
Int

```swift
let disposable = textField.rx.text.orEmpty
    .map { $0.count }
    .filter { $0 > 3 }
    .subscribe(onNext: {
        print("\($0)")
    })

    …

    disposable.dispose()
```

```swift
class MyViewController : UIViewController {
    let disposeBag = DisposeBag()

    override func viewDidLoad() {
        textField.rx.text.orEmpty
            .map { $0.count }
            .filter { $0 > 3 }
            .subscribe(onNext: {
                print("\($0)")
            })
            .disposed(by: disposeBag)
    }
}
```


1:59
Giphy
Search


rx.text
GiphySearch
filter
debounce
distinctUntilChanged
flatMap
map
observeOn
subscribe

```
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
```

```
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
```

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
```

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
```

```swift
extension URLSession {
    public func data(request: URLRequest) -> Observable<Data> {
        return Observable.create { observer in
            let task = URLSession.shared.dataTask(with: request) { (data, response, error) in
                guard let response = response, let data = data else {
                    observer.on(.error(error ?? RxDemoURLError.unknown))
                    return
                }

                guard let httpResponse = response as? HTTPURLResponse else {
                    observer.on(.error(RxDemoURLError.nonHTTPResponse(response: response)))
                    return
                }

                guard 200 ..< 300 ~= httpResponse.statusCode else {
                    observer.on(.error(RxDemoURLError.httpRequestFailed(response: httpResponse)))
                    return
                }

                observer.on(.next(data))
                observer.on(.completed)
            }

            task.resume()

            return Disposables.create(with: task.cancel)
        }
    }
}
```

```swift
extension URLSession {
    public func data(request: URLRequest) -> Observable<Data> {
        return Observable.create { observer in
            let task = URLSession.shared.dataTask(with: request) { (data, response, error) in
                guard let response = response, let data = data else {
                    observer.on(.error(error ?? RxDemoURLError.unknown))
                    return
                }

                guard let httpResponse = response as? HTTPURLResponse else {
                    observer.on(.error(RxDemoURLError.nonHTTPResponse(response: response)))
                    return
                }

                guard 200 ..< 300 ~= httpResponse.statusCode else {
                    observer.on(.error(RxDemoURLError.httpRequestFailed(response: httpResponse)))
                    return
                }

                observer.on(.next(data))
                observer.on(.completed)
            }

            task.resume()

            return Disposables.create(with: task.cancel)
        }
    }
}
```

```swift
extension URLSession {
    public func data(request: URLRequest) -> Observable<Data> {
        return Observable.create { observer in
            let task = URLSession.shared.dataTask(with: request) { (data, response, error) in
                guard let response = response, let data = data else {
                    observer.on(.error(error ?? RxDemoURLError.unknown))
                    return
                }

                guard let httpResponse = response as? HTTPURLResponse else {
                    observer.on(.error(RxDemoURLError.nonHTTPResponse(response: response)))
                    return
                }

                guard 200 ..< 300 ~= httpResponse.statusCode else {
                    observer.on(.error(RxDemoURLError.httpRequestFailed(response: httpResponse)))
                    return
                }

                observer.on(.next(data))
                observer.on(.completed)
            }

            task.resume()

            return Disposables.create(with: task.cancel)
        }
    }
}
```

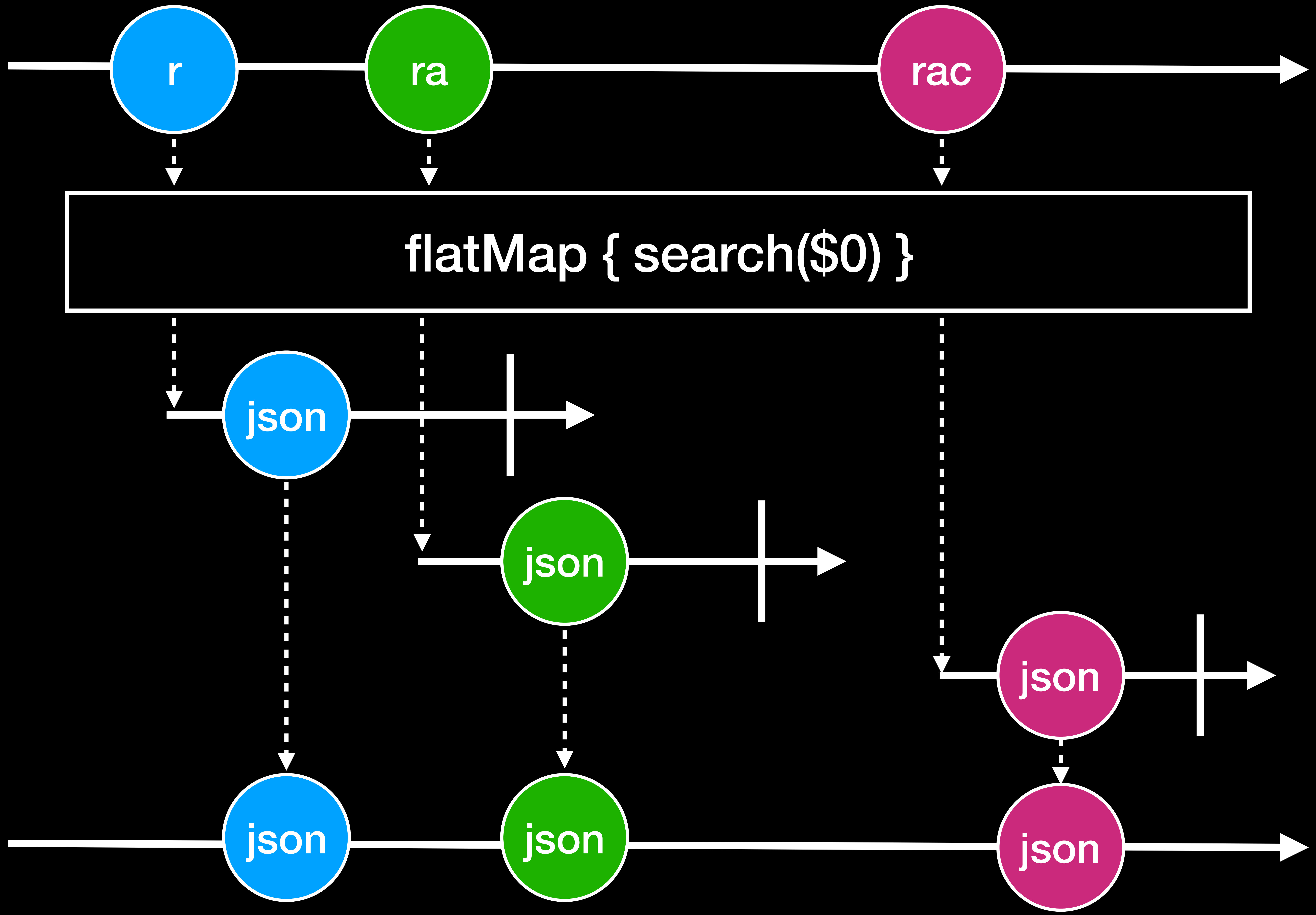
```swift
extension URLSession {
    public func data(request: URLRequest) -> Observable<Data> {
        return Observable.create { observer in
            let task = URLSession.shared.dataTask(with: request) { (data, response, error) in
                guard let response = response, let data = data else {
                    observer.on(.error(error ?? RxDemoURLError.unknown))
                    return
                }

                guard let httpResponse = response as? HTTPURLResponse else {
                    observer.on(.error(RxDemoURLError.nonHTTPResponse(response: response)))
                    return
                }

                guard 200 ..< 300 ~= httpResponse.statusCode else {
                    observer.on(.error(RxDemoURLError.httpRequestFailed(response: httpResponse)))
                    return
                }

                observer.on(.next(data))
                observer.on(.completed)
            }

            task.resume()

            return Disposables.create(with: task.cancel)
        }
    }
}
```

```swift
extension URLSession {
    public func data(request: URLRequest) -> Observable<Data> {
        return Observable.create { observer in
            let task = URLSession.shared.dataTask(with: request) { (data, response, error) in
                guard let response = response, let data = data else {
                    observer.on(.error(error ?? RxDemoURLError.unknown))
                    return
                }

                guard let httpResponse = response as? HTTPURLResponse else {
                    observer.on(.error(RxDemoURLError.nonHTTPResponse(response: response)))
                    return
                }

                guard 200 ..< 300 ~= httpResponse.statusCode else {
                    observer.on(.error(RxDemoURLError.httpRequestFailed(response: httpResponse)))
                    return
                }

                observer.on(.next(data))
                observer.on(.completed)
            }

            task.resume()

            return Disposables.create(with: task.cancel)
        }
    }
}
```

```swift
extension URLSession {
    public func data(request: URLRequest) -> Observable<Data> {
        return Observable.create { observer in
            let task = URLSession.shared.dataTask(with: request) { (data, response, error) in
                guard let response = response, let data = data else {
                    observer.on(.error(error ?? RxDemoURLError.unknown))
                    return
                }

                guard let httpResponse = response as? HTTPURLResponse else {
                    observer.on(.error(RxDemoURLError.nonHTTPResponse(response: response)))
                    return
                }

                guard 200 ..< 300 ~= httpResponse.statusCode else {
                    observer.on(.error(RxDemoURLError.httpRequestFailed(response: httpResponse)))
                    return
                }

                observer.on(.next(data))
                observer.on(.completed)
            }

            task.resume()

            return Disposables.create(with: task.cancel)
        }
    }
}
```

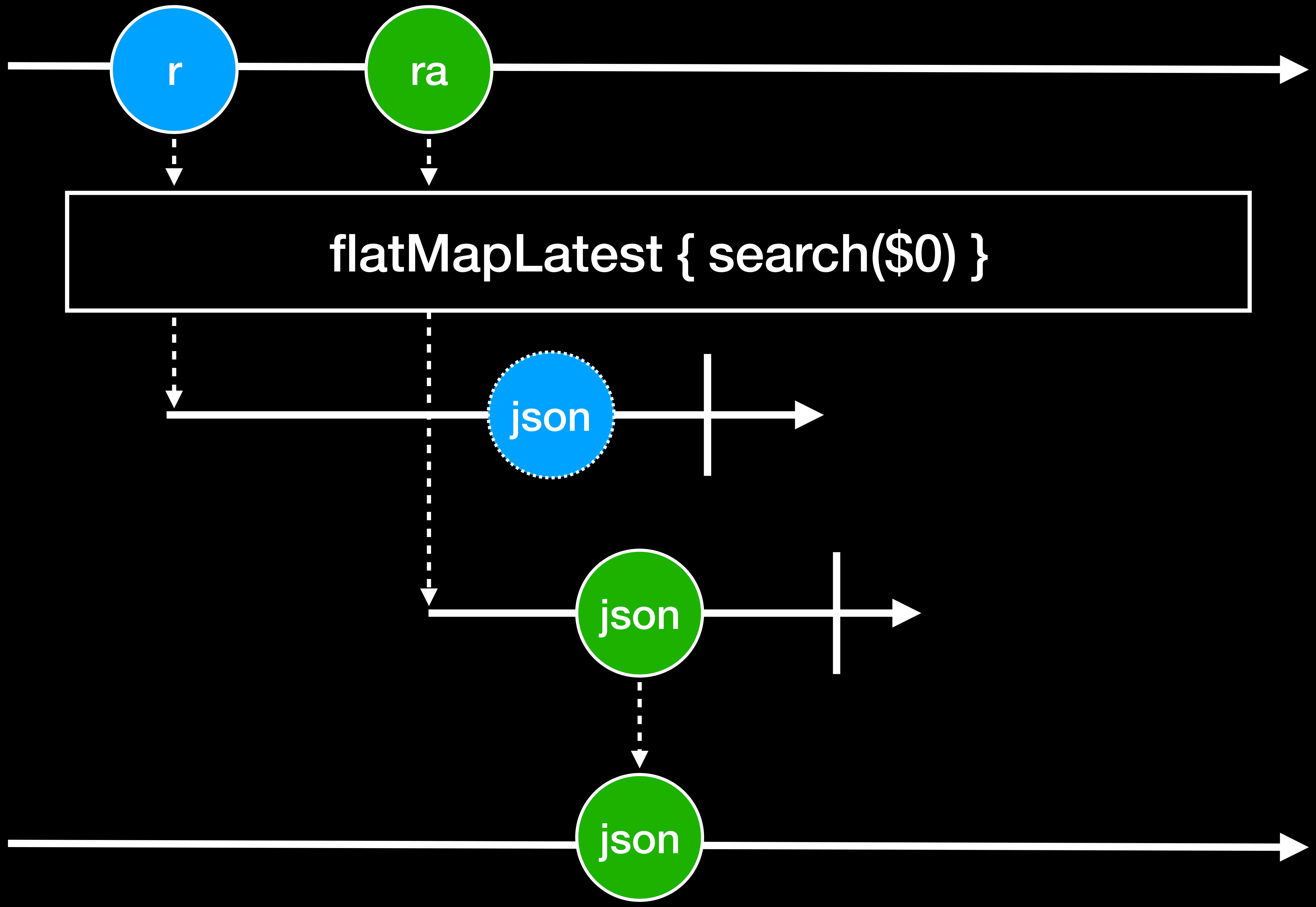
```
extension URLSession {
    public func data(request: URLRequest) -> Observable<Data> {
        return Observable.create { observer in
            let task = URLSession.shared.dataTask(with: request) { (data, response, error) in
                guard let response = response, let data = data else {
                    observer.on(.error(error ?? RxDemoURLError.unknown))
                    return
                }

                guard let httpResponse = response as? HTTPURLResponse else {
                    observer.on(.error(RxDemoURLError.nonHTTPResponse(response: response)))
                    return
                }

                guard 200 ..< 300 ~= httpResponse.statusCode else {
                    observer.on(.error(RxDemoURLError.httpRequestFailed(response: httpResponse)))
                    return
                }

                observer.on(.next(data))
                observer.on(.completed)
            }

            task.resume()

            return Disposables.create(with: task.cancel)
        }
    }
}
```

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .map { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
    }
```


r
ra
rac
flatMap { search($0) }
json
json
json
json
json
json

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMap { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
    }
```

```
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
    }
```


r
ra
flatMapLatest { search($0) }
json
json
json

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
    }
    .map { self.parseJSONResults($0) }
    .map { self.parseRemoteModels($0) }
```

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
    }
    .map { self.parseJSONResults($0) }
    .map { self.parseRemoteModels($0) }
    .subscribe(onNext: {
        self.gifs = $0.compactMap(GifModel.init)
        self.collectionView.reloadData()
    })
```

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
            .retry(3)
            .catchError { _ in Observable.empty() }
    }
    .map { self.parseJSONResults($0) }
    .map { self.parseRemoteModels($0) }
    .catchErrorJustReturn([])
    .subscribe(onNext: {
        self.gifs = $0.compactMap(GifModel.init)
        self.collectionView.reloadData()
    })
```

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
            .retry(3)
            .catchError { _ in Observable.empty() }
    }
    .map { self.parseJSONResults($0) }
    .map { self.parseRemoteModels($0) }
    .catchErrorJustReturn([])
    .subscribe(onNext: {
        self.gifs = $0.compactMap(GifModel.init)
        self.collectionView.reloadData()
    })
```

```
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
            .retry(3)
            .catchError { _ in Observable.empty() }
    }
    .map { self.parseJSONResults($0) }
    .map { self.parseRemoteModels($0) }
    .catchErrorJustReturn([])
    .subscribe(onNext: {
        self.gifs = $0.compactMap(GifModel.init)
        self.collectionView.reloadData()
    })
```

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .observeOn(ConcurrentDispatchQueueScheduler(qos: .background))
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
            .retry(3)
            .catchError { _ in Observable.empty() }
    }
    .map { self.parseJSONResults($0) }
    .map { self.parseRemoteModels($0) }
    .catchErrorJustReturn([])
    .observeOn(MainScheduler.instance)
    .subscribe(onNext: {
        self.gifs = $0.compactMap(GifModel.init)
        self.collectionView.reloadData()
    })
```

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .observeOn(ConcurrentDispatchQueueScheduler(qos: .background))
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
            .retry(3)
            .catchError { _ in Observable.empty() }
    }
    .map { self.parseJSONResults($0) }
    .map { self.parseRemoteModels($0) }
    .catchErrorJustReturn([])
    .observeOn(MainScheduler.instance)
    .subscribe(onNext: {
        self.gifs = $0.compactMap(GifModel.init)
        self.collectionView.reloadData()
    })
```

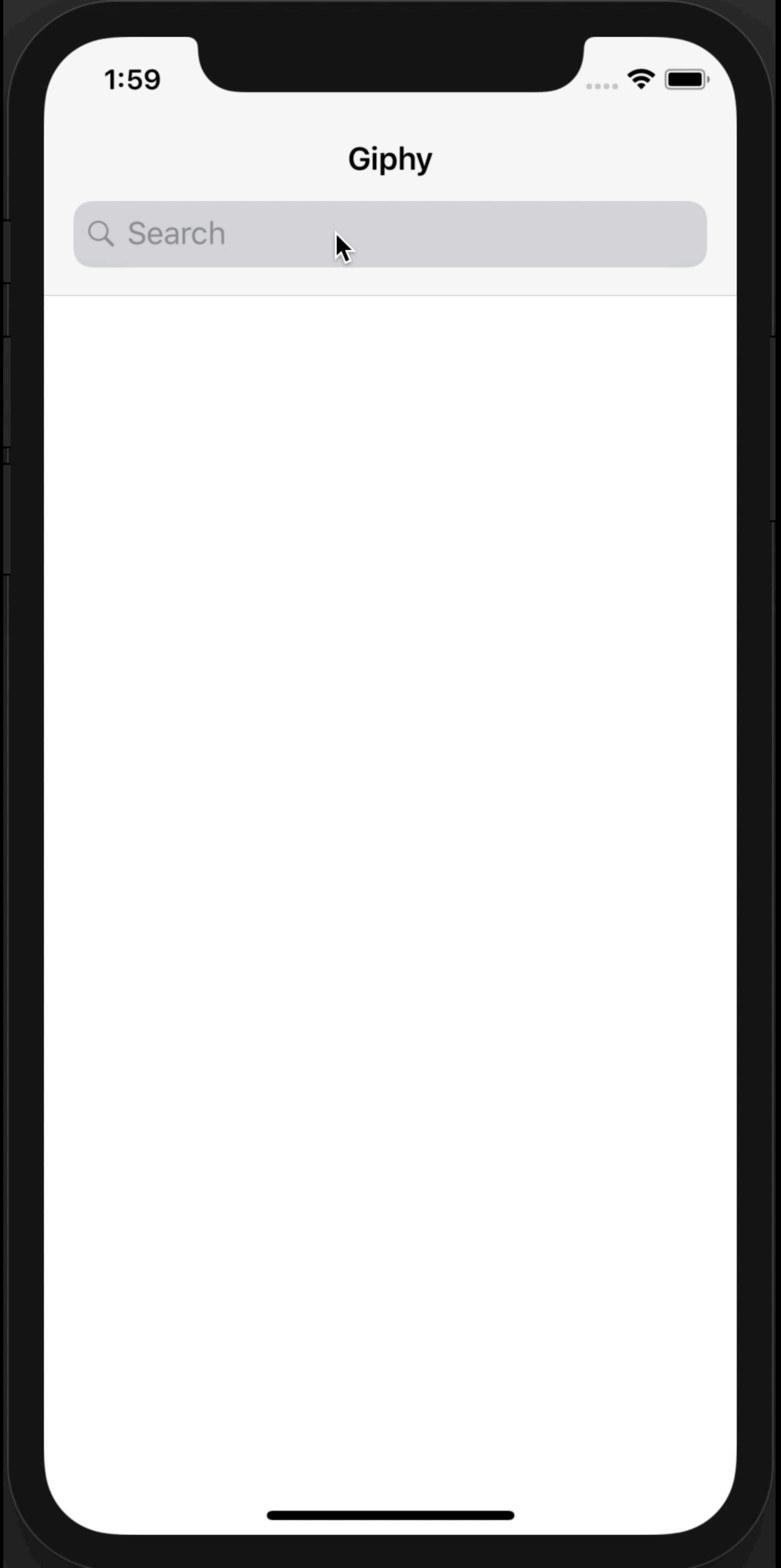
```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .observeOn(ConcurrentDispatchQueueScheduler(qos: .background))
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
            .retry(3)
            .catchError { _ in Observable.empty() }
    }
    .map { self.parseJSONResults($0) }
    .map { self.parseRemoteModels($0) }
    .catchErrorJustReturn([])
    .observeOn(MainScheduler.instance)
    .subscribe(onNext: {
        self.gifs = $0.compactMap(GifModel.init)
        self.collectionView.reloadData()
    })
```

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .do(onNext: { _ in
        UIApplication.shared.isNetworkActivityIndicatorVisible = true
    })
    .observeOn(ConcurrentDispatchQueueScheduler(qos: .background))
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
            .retry(3)
            .catchError { _ in Observable.empty() }
    }
    .map { self.parseJSONResults($0) }
    .map { self.parseRemoteModels($0) }
    .catchErrorJustReturn([])
    .observeOn(MainScheduler.instance)
    .do(onNext: { _ in
        UIApplication.shared.isNetworkActivityIndicatorVisible = false
    })
    .subscribe(onNext: {
        self.gifs = $0.compactMap(GifModel.init)
        self.collectionView.reloadData()
    })
```

```
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .do(onNext: { _ in
        UIApplication.shared.isNetworkActivityIndicatorVisible = true
    })
    .observeOn(ConcurrentDispatchQueueScheduler(qos: .background))
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
            .retry(3)
            .catchError { _ in Observable.empty() }
    }
    .map { self.parseJSONResults($0) }
    .map { self.parseRemoteModels($0) }
    .catchErrorJustReturn([])
    .observeOn(MainScheduler.instance)
    .do(onNext: { _ in
        UIApplication.shared.isNetworkActivityIndicatorVisible = false
    })
    .subscribe(onNext: {
        self.gifs = $0.compactMap(GifModel.init)
        self.collectionView.reloadData()
    })
```

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .do(onNext: { _ in
        UIApplication.shared.isNetworkActivityIndicatorVisible = true
    })
    .observeOn(ConcurrentDispatchQueueScheduler(qos: .background))
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
            .retry(3)
            .catchError { _ in Observable.empty() }
    }
    .map { self.parseJSONResults($0) }
    .map { self.parseRemoteModels($0) }
    .catchErrorJustReturn([])
    .observeOn(MainScheduler.instance)
    .do(onNext: { _ in
        UIApplication.shared.isNetworkActivityIndicatorVisible = false
    })
    .subscribe(onNext: {
        self.gifs = $0.compactMap(GifModel.init)
        self.collectionView.reloadData()
    })
```

```swift
searchBar.rx.text
    .orEmpty
    .filter { $0.count > 1 }
    .debounce(0.5, scheduler: MainScheduler.instance)
    .distinctUntilChanged()
    .do(onNext: { _ in
        UIApplication.shared.isNetworkActivityIndicatorVisible = true
    })
    .observeOn(ConcurrentDispatchQueueScheduler(qos: .background))
    .flatMapLatest { query -> Observable<Data> in
        let url = URL(string: "https://api.giphy.com/v1/gifs/search?&q=\(query)")!
        let request = URLRequest(url: url)
        return URLSession.shared.data(request: request)
            .retry(3)
            .catchError { _ in Observable.empty() }
    }
    .map { self.parseJSONResults($0) }
    .map { self.parseRemoteModels($0) }
    .catchErrorJustReturn([])
    .observeOn(MainScheduler.instance)
    .do(onNext: { _ in
        UIApplication.shared.isNetworkActivityIndicatorVisible = false
    })
    .subscribe(onNext: {
        self.gifs = $0.compactMap(GifModel.init)
        self.collectionView.reloadData()
    })
```


1:59
Giphy
Search

Final Thoughts

Some Advice

- Experiment with Rx using a playground
- Learn operators @ http://rxmarbles.com/
- Break down your chain into smaller observables
- Use Rx `debug` operators

More Advice

- Using lazy vars with observables can create retain cycles
- Prefer `*weak*` over `*unowned*` when capturing *self* to avoid crashes
- Prefer capturing variables over *self* when possible to reduce retain cycles
- Avoid side effects during disposal

Pros

- Less code
- Maintainable code
- Readable code
- Eliminate race conditions
- Improve programming skills

Cons

- Debugging can be difficult with long stack traces
- Steep learning curve
- Everything starts looking like an observable!

Resources

- https://github.com/ReactiveX/RxSwift
- http://reactivex.io/
- https://github.com/RxSwiftCommunity
- https://rxswift.slack.com
- http://github.com/c0diq/RxDemo

Questions?