RxCocoa

✅ Intro

Binder
- a useful construct which represents something that can accept new values, but can't be subscribed to
- let you bind values into some specific implementation or underlying object.
- it can't accept errors
- it also takes care of weakifying and retaining its base object
  - so you don't have to deal with pesky memory leaks or weak references.

Decodable
- conforming struct which will be used as a data model to correctly map the JSON response to something more easily digested by Swift:

✅ RxCocoa with basic UIKit Tools

- **Display data using RxCocoa:**

Observables are entities capable of notifying subscribers that some data has arrived or changed, pushing values to be processed.
- the correct place to subscribe to an observable while working in view controllers is inside **viewDidLoad**
  - this is because *you need to subscribe as early as possible, but only after the view has been loaded.*
  - Subscribing in a different lifecycle event might lead to missed events, duplicate subscriptions, or parts of the UI that might be visible before you bind data to them.

  you have to create all subscriptions before the application creates or requests data that needs to be processed and displayed to the user

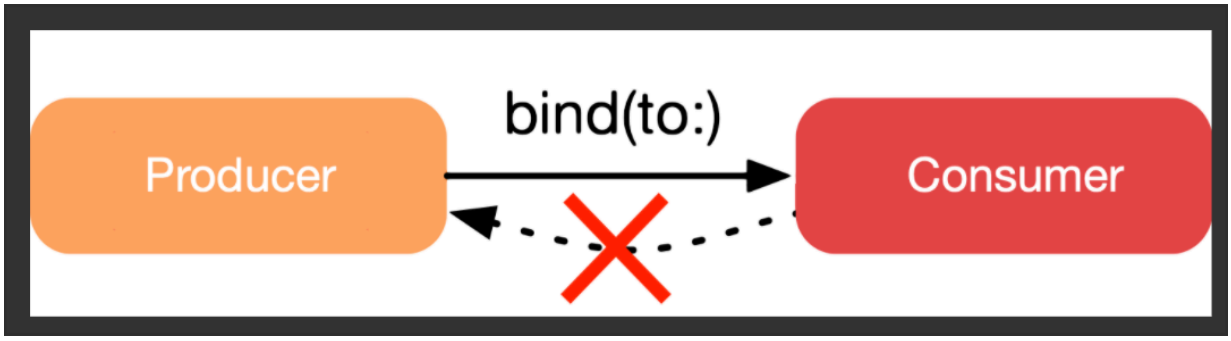
- subscription should be canceled when the view controller is dismissed to avoid potential memory leaks

```swift
override func viewDidLayoutSubviews() {
  super.viewDidLayoutSubviews()

  Appearance.applyBottomLine(to: searchCityName)

    ApiController.shared.currentWeather(for: "RxSwift")
       //observeOn(): wrap source sequence to run its observer
           callback in Main scheduler
        .observeOn(MainScheduler.instance)
        .subscribe(onNext: {data in
              self.tempLabel.text = "\(data.temperature)° C"
              self.iconLabel.text = data.icon
              self.humidityLabel.text = "\(data.humidity)%"
              self.cityNameLabel.text = data.cityName
        })
        .disposed(by: bag)
```

```swift
searchCityName.rx.text.orEmpty
  // rx.text: return ControlProperty<String?> which
  //          conform to both Observable & Observer type
  //      => you can subscribe to it and also add new value onto
      it to set the text field
  // orEmty : transform ControlProperty<String?> to
      ControlProperty<String>. Emits an empty string if nil is
      emitted
  //      <=> equivalent of .map { $0 ?? "" }
    .filter{ !$0.isEmpty} // filter out empty value
    .flatMap{text in
        /// **fetch data from api and return a new observable with the data to
            display
        ApiController.shared
            .currentWeather(for: text)
            .catchErrorJustReturn(.empty)
    }
    .observeOn(MainScheduler.instance) ///switch to correct thread & display
        updated data
    .subscribe(onNext: { data in
        self.tempLabel.text = "\(data.temperature)° C"
        self.iconLabel.text = data.icon
        self.humidityLabel.text = "\(data.humidity)%"
        self.cityNameLabel.text = data.cityName
    })
    .disposed(by: bag)
```

- **Binding Observable :**

- In RxCocoa, a binding is a **uni-directional** stream of data



    - A **producer**, which produces the value.
    - A **consumer**, which processes the values from the producer.
- A consumer cannot return a value. This is a general rule when using bindings in RxSwift.

- **bind(to:)** is used to bind an observable to another entity.
    - It's required that the consumer conforms to ObserverType, a write-only entity that can only accept new events but cannot be subscribed to
        - the only ObserverType in RxSwift is Subject
    - **bind(to:)** is an **alias** for **subscribe**():
        - Calling bind(to: observer) will *internally call subscribe(observer)*.

- Using binding observable to display data:

EX: we can replace the searchCityName code above to:

```
let search = searchCityName.rx.text.orEmpty
  // rx.text: return ControlProperty<String?> which
  //          conform to both Observable & Observer type
  //      => you can subscribe to it and also add new value onto it to set the text field
  // orEmty : transform ControlProperty<String?> to ControlProperty<String>. Emits an empty string if nil is emitted
  //      <=> equivalent of .map { $0 ?? "" }
    .filter{ !$0.isEmpty} // filter out empty value
/**
    .flatMap{text in
        /// **fetch data from api and return a new observable with the data to display
        ApiController.shared
            .currentWeather(for: text)
            .catchErrorJustReturn(.empty)
    }

    .observeOn(MainScheduler.instance) ///switch to correct thread & display updated data

    .subscribe(onNext: { data in
        self.tempLabel.text = "\(data.temperature)° C"
        self.iconLabel.text = data.icon
        self.humidityLabel.text = "\(data.humidity)%"
        self.cityNameLabel.text = data.cityName
    })
    .disposed(by: bag)
 **/
```

```
      // we can replace this part of code to:
      //
        .flatMapLatest{ text in
            /// **
            /// fetch data from api and return a new observable with the data to display
            /// will cancel any previous network requests when a new one starts.
            ApiController.shared
                .currentWeather(for: text)
                .catchErrorJustReturn(.empty)
        }
            ///  ** share(replay:_) :**makes your stream reusable and transforms a single-use data source into a multi-use Observable
        .share(replay: 1)
        .observeOn(MainScheduler.instance)
```

```
 // With the change above 👆
 //   + it's possible to process every single parameter from a different subscription, mapping the value required to be displayed
 /// EX: get the temperature as a string out of the shared data source observable
///
search.map(\.icon)
  .bind(to: iconLabel.rx.text)
  .disposed(by: bag)

search.map { "\($0.humidity)%" }
  .bind(to: humidityLabel.rx.text)
  .disposed(by: bag)

search.map(\.cityName)
  .bind(to: cityNameLabel.rx.text)
  .disposed(by: bag)
```

🟡 Improving code with Traits

- **Traits:**
  - specialized implementations of observables
  - **a group of ObservableType-conforming objects,** which are specialized for creating straightforward, easy-to-write code, especially when working with UI.

- Rx Cocoa traits are:
  - ControlProperty and ControlEvent
  - Driver
  - Signal

- **Trait's rules:**
  - They can’t error out.
  - They are observed and subscribed on the main scheduler.
  - They share resources since they are both derived from an entity called SharedSequence.
    - Driver automatically gets share(replay: 1), while Signal gets share().

=> This ensures something is always displayed in the user interface and that they are always able to be handled by the user interface.

- ControlProperty: bind the data to the correct user interface component using the dedicated '**rx'** namespace

- ControlEvent: listen for a certain event of the UI component such as Button Pressed
  - A control event is available if the component uses UIControl.Events to keep track of its current status.

- Driver: a special observable with the same constraints as explained before, so it can’t error out.
  - All processes are ensured to execute on the main thread, which avoids making UI changes on background threads
  - it always shares resources and replays its latest value to new consumers upon subscription.
  - Driver is more suitable for modeling state, due to their different replay strategies.

- Signal:
  - delivers events on the main scheduler, doesn’t error out, and shares its resources
  - BUT it doesn’t replay its latest value to new consumers upon

subscription.
    - useful for modeling events

- **Improve code with ControlProperty & Driver:**

– It's easy to forget to call:
.observeOn(MainScheduler.instance)
and end up creating UI processes on a background thread.

=> Therefore we can **transform the object observable to Driver** with **asDriver(onError...:_)**

Ex: we can replace the observeOn() of 'search' to :

```
let search = searchCityName.rx.text.orEmpty
  .filter { !$0.isEmpty }
  .flatMapLatest { text in
    ApiController.shared
      .currentWeather(for: text)
      .catchErrorJustReturn(.empty)
  }
  .asDriver(onErrorJustReturn: .empty)
```

=> There are different variants of asDrive() to converts observable in to a Driver:

- **asDriver(onErrorJustReturn:):**
    - the *onErrorJustReturn* parameter specifies a default value to be used in case the converted observable errors out

  => eliminating the possibility for the driver itself to emit an error.

- **asDriver(onErrorDriveWith:):**
    - you can handle the error manually and return a new Driver generated for this purpose only.

- **asDriver(onErrorRecover:):**

- Can be used alongside another existing Driver.
- This will come in play to recover the current Driver that just encountered an error.

- **NOTE: When we turn observable to a driver, we also need to replace **bind(to: _)** to **drive()**

```
search.map(\.icon)
  //.bind(to: iconLabel.rx.text)       // as we convert observable to Driver, we need to use drive() instead
  .drive(iconLabel.rx.text)
  .disposed(by: bag)

search.map { "\($0.humidity)%" }
  //.bind(to: humidityLabel.rx.text)   // as we convert observable to Driver, we need to use drive() instead
  .drive(humidityLabel.rx.text)
  .disposed(by: bag)

search.map(\.cityName)
  //.bind(to: cityNameLabel.rx.text)   // as we convert observable to Driver, we need to use drive() instead
  .drive(cityNameLabel.rx.text)
  .disposed(by: bag)
```

- And we can change this part

```
let search = searchCityName.rx.text.orEmpty
```

to:

```
let search = searchCityName.rx
  .controlEvent(.editingDidEndOnExit)
  .map { self.searchCityName.text ?? "" }
  // rest of your .filter { }.flatMapLatest { } continues here
```

  - The old code call a lot of unescessary API requests while typing city's name.
  - Now the application retrieves the weather only when the user hits the **“Search”** button on the keyboard.

=> You’re not making unnecessary network requests, and the code is controlled at compile time by Traits.

- **Traits in RxSwift & in RxCocoa:**

Name	Events			Shares Effect	Replay	Scheduler
	next	error	complete			
Observable	✔	✔	✔	✘	✘	Any
PublishSubject	✔	✔	✔	✘	✘	Any
BehaviorSubject	✔	✔	✔	✘	✔	Any
Driver	✔	✘	✔	✔	✔	**Main**
Signal	✔	✘	✔	✔	✘	**Main**
Completable	✘	✔	✔	✔	N/A	Any
Single	✔ **(1)**	✔	✘	✘	✘	Any
Maybe	✔ **(1)**	✔	✔	✘	✘	Any
BehaviorRelay	✔	✘	✘	✘	✔	Any
PublishRelay	✔	✘	✘	✘	✘	Any

🟡 Disposing with RxCocoa

- There's a ***bag*** inside the main view controller that takes care of disposing all the subscriptions when the view controller is deallocated.

- **Unowned vs weak with RxCocoa**
  - Using weak and unowned are the same you would follow when using regular Swift closures.
    - calling the closure-variations of Rx, such as subscribe(onNext:)
  - If your closure is an escaping closure, use either a weak or unowned capture group;
    - otherwise, you might get a retain cycle and your subscriptions will never be released
  - Using *weak* means you'll get an Optional reference to **self.**
    - weak providing **Self?**
  - Using *unowned* will provide an implicitly unwrapped reference to **self**.
    - unowned providing **Self!**

- it’s practically a force-unwrap; if the object isn’t there, your app will crash.

🟡 UI with RxCocoa

Ref Link : https://medium.com/@VincentVuVNG/d%C3%B9ng-th%C6%B0-vi%E1%BB%87n-rxswift-%C4%91%E1%BB%83-c%E1%BA%A3i-ti%E1%BA%BFn-hi%E1%BB%87u-su%E1%BA%A5t-cho-d%E1%BB%B1-%C3%A1n-ios-c%E1%BB%A7a-b%E1%BA%A1n-ph%E1%BA%A7n-5-56470ce79e57

- Button : onClick

```
btn.rx.controlEvent(UIControlEvents.touchUpInside).subscribe {
(event) in
    print("Btn Clicked")
}
btn.rx.tap.bind { (event) in
    print("Btn Clicked")
}
```

- TextField: text change

```
tf.rx.text.asObservable().subscribe(onNext: { (text) in
        guard let textGet = text else {
            return
        }
        print("Tf change : \(textGet)")
 })
```

- ScrollView: offset , didScroll

```
scrollView.rx.contentOffset.subscribe(onNext: { (point) in
    print("Scroll offset : \(point)")
})
scrollView.rx.didScroll.subscribe(onNext: { _ in
     print("Did scroll : \(self.scrollView.contentOffset)")
})
```

- Slider:

```
scrollView.rx.contentOffset.subscribe(onNext: { (point) in
    print("Scroll offset : \(point)")
})
scrollView.rx.didScroll.subscribe(onNext: { _ in
     print("Did scroll : \(self.scrollView.contentOffset)")
})
```

- Gesture:

```
let tap = UITapGestureRecognizer()
self.view.addGestureRecognizer(tap)
tap.rx.event.asObservable().subscribe(onNext: { (tapGestures) in
     print("Tap Gesture tap")
})
```

- Method:

```
//Ví dụ trong UIViewController đợi viewDidAppear


self.rx.methodInvoked(#selector(self.viewDidAppear(_:))).subscribe(o
nNext: { (event) in
   print("View did Appear")
})
```

- TableView:

```
// Tạo 1 model
struct ModelDetail {
    var title:String = ""
}


// Register cell
self.table.register(UINib(nibName: "MyCell", bundle: nil),
forCellReuseIdentifier: "MyCell")


// Tạo data source
let items = Observable.of([
ModelDetail(title:"Item1"),
ModelDetail(title:"Item2"),
ModelDetail(title:"Item3"),
ModelDetail(title:"Item4")])


// Binding
items.bind(to: self.table.rx.items(cellIdentifier: "MyCell",
cellType: MyCell.self)){ (index,model,cell) in
    cell.lbl.text = model.title
}.disposed(by: dispose)
```

Table selected indexPath:

```
// Did Selected Indexpath
self.table.rx.itemSelected.asObservable().subscribe(onNext: {
(indexPath) in
    print("Row Selected :\(indexPath.row)")
})
.disposed(by: dispose)
```

- CollectionView:

```
// Binding

items.bind(to: collectionView.rx.items) { (collectionView, row,
element) in
        let indexPath = IndexPath(row: row, section: 0)
        let cell =
collectionView.dequeueReusableCell(withReuseIdentifier: "MyCell",
for: indexPath) as! MyCell
        cell.lbl?.text = element.title
        return cell
    }
    .disposed(by: dispose)


// Selected IndexPath
self.collectionView.rx.itemSelected.subscribe(onNext: { (indexPath)
in
    print("CollectionView Row Selected :\(indexPath.row)")
})
.disposed(by: dispose)
```

Custom delegate along with RX

```
// set delegate
table.rx.setDelegate(self).disposed(by: dispose)
collectionView.rx.setDelegate(self).disposed(by: dispose)


// implement delegate
extension ViewController : UITableViewDelegate {
    func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {
    }
}
extension ViewController : UICollectionViewDelegate {
    func collectionView(_ collectionView: UICollectionView,
didSelectItemAt indexPath: IndexPath) {
    }
}
```