Introduction

Outline

1. What is RxCocoa?
2. Why using RxCocoa?
3. Traditional vs Reactive

Sun*

1. What is RxCocoa?

- **RxSwift** library allows us to use **Swift** disparately. With this library, asynchronous programming becomes easier to do and more legible. It allows you to build more solid architectures and applications with higher quality.

- **RxCocoa** is a standalone library (though it's bundled with **RxSwift**) that allows you to use many prebuilt features to integrate better with UIKit and Cocoa.

2. Why using RxCocoa?

- **RxCocoa** provides extensions to the Cocoa and Cocoa Touch frameworks to take advantage of RxSwift.

- **RxCocoa** enables you to do reactive networking, react to user interactions, bind data models to UI controls, and more.

3. Traditional vs Reactive

- Let’s consider this ViewModel:

```
import Foundation

struct PeopleViewModel {
    let data = [
        "Ben Sandofsky",
        "Carla White",
        "Jaimee Newberry",
        "Natasha Murashev",
        "Robi Ganguly"
    ]
}
```

3. Traditional vs Reactive

- With traditional way, we have to implement lots of things to display data to a **UITableView**

```
class PeopleViewController: UIViewController, UITableViewDataSource, UITableViewDelegate {

    @IBOutlet private weak var peopleTableView: UITableView!

    let peopleViewModel = PeopleViewModel()

    override func viewDidLoad() {
        super.viewDidLoad()
        peopleTableView.dataSource = self
        peopleTableView.delegate = self
    }

    func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return peopleViewModel.data.count
    }

    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: IndexPath) -> UITableViewCell {
        guard let cell = tableView.dequeueReusableCellWithIdentifier("PersonCell")
            else {
                return UITableViewCell()
        }

        cell.configCell(name: peopleViewModel.data[indexPath.row])
        return cell
    }

    func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: IndexPath) {
        print("You selected \(peopleViewModel.data[indexPath.row])")
    }
}
```

3. Traditional vs Reactive

- Let’s modify this ViewModel in Reactive way:

```
import RxSwift
import RxCocoa

struct PeopleViewModel {

    let data = Driver.just([
        "Ben Sandofsky",
        "Carla White",
        "Jaimee Newberry",
        "Natasha Murashev",
        "Robi Ganguly",
        "Virginia Roberts",
        "Scott Gardner"
    ])
}
```

3. Traditional vs Reactive

- With Rx, you don't have to implement too much boilerplate code

```swift
import UIKit
import RxSwift
import RxCocoa

class ViewController: UIViewController {

    @IBOutlet private weak var peopleTableView: UITableView!

    let peopleViewModel = PeopleViewModel()
    let disposeBag = DisposeBag()

    override func viewDidLoad() {
        super.viewDidLoad()

        peopleTableView.register(UITableViewCell.self, forCellReuseIdentifier: "Cell")

        peopleViewModel.data
            .drive(peopleTableView.rx.items) { tableView, index, item in
                guard let cell = tableView.dequeueReusableCell(withIdentifier: "Cell") else {
                    return UITableViewCell()
                }
                cell.textLabel?.text = item
                return cell
            }
            .disposed(by: disposeBag)

        peopleTableView.rx.modelSelected(String.self)
            .subscribe(onNext: {
                print("You selected \($0)")
            })
            .disposed(by: disposeBag)
    }
}
```

3. Traditional vs Reactive

- With Reactive, you write 40% less code to do the exact same thing.

- Additionally, it’s not just about writing less code. It’s about writing more expensive ocde and doing more with that code, especially when it comes to writing asynchronous code

Question & Answer?