Урок 5

RxSwift

Реактивные биндинги и управление сигналами

RxSwift

Практическое задание

Дополнительные материалы

Используемая литература

RxSwift

На этом уроке мы рассмотрим библиотеку **RxSwift.** Существует огромное количество различных библиотек, но именно эта предлагает не просто решение проблемы, а абсолютно другой подход к разработке – функционально-реактивное программирование.

Эта библиотека обязательна для изучения, но не для использования. Её поклонников так же много, как и противников. У неё много плюсов и минусов. Она позволяет очень просто делать некоторые вещи, но при этом её очень сложно понять до конца.

RxSwift стоит на трех столпах: источниках данных, модификаторах и слушателях. Представьте себе игру в испорченный телефон. Первый человек говорит какое-то слово, стоящие за ним люди слушают его и передают друг другу, в конце находится последний человек, которому передают изначальное слово в искажённом виде – часто настолько искажённом, что оно совершенно не похоже на изначальный вариант.

В этой схеме первый человек – это источник данных, последний слушатель – получатель информации, а люди в середине цепочки – модификаторы.

В отличие от примера с испорченным телефоном источник может передавать любые данные. Модификаторы искажают первоначальную информацию не случайно, а по определённому алгоритму с определенной целью.

Более того, вы уже видели нечто похожее на этот алгоритм: это модификация массива с помощью **map:**

```
let source = [0, 1, 2, 3]
let recipient                    // Получатель
    = source                     // Источник
        .map { $0 * 2 }          // Модификатор 1
        .reduce(0, { $0 + $1 }) // Модификатор 2
```

Это очень похоже на **RxSwift**, но у неё больше возможностей.

В роли **Observable** может выступать множество различных элементов кода – переменная, функция, свойство. Мы можем создать **Observable** даже из константы.

```
Observable
            .just(100)             // Создаём событие
            .subscribe { event in // Добавляем слушателя
                print(event)       // Выводим в консоль событие
        }
```

Первый пример. Используя статический метод **just** класса **Observable**, мы создаём самый простой источник событий. Далее с помощью метода **subscribe** мы регистрируем замыкание слушателя. Он выводит сообщения в консоль. Вывод будет выглядеть вот так.

```
next(100)
completed
```

В примере мы видим два события:

- next – главное событие, которое генерирует источник, оно несёт полезную нагрузку.;

- complete – случайное событие, которое информирует, что источник завершил работу и больше событий не будет.

Сначала мы получили событие с информацией 100 – именно это делает метод **just**. Он берёт одно значение и генерирует одно событие с этим значением, после чего источник завершает работу.

Мы много раз говорили о том, что источник генерирует последовательность. Это значит, что он может отправлять несколько событий по очереди:

```
Observable<Int>
            .interval(1, scheduler: MainScheduler.instance)
            .subscribe { e in
                print(e)
        }
```

В примере **Observable**, созданный с помощью метода **interval**, генерирует новое событие с указанной периодичностью, в данном случае с интервалом в 1 секунду. **MainScheduller** указывает, что генерироваться события будут на главной очереди. Такая последовательность никогда не завершится, то есть мы никогда не получим событие **complete.**

Забрасывать цифры в замыкания, конечно, весело, но бессмысленно. Давайте начнём с UI-биндингов. Что это такое? В библиотеке есть расширение классов **UIKit.** Они добавляют свойства к различным UI-элементам, на которые можно подписываться.

Модифицируем экран авторизации. Добавим импорт двух библиотек:

```
import RxSwift
import RxCocoa
```

Протянем **IBOutlet** от кнопки:

```
@IBOutlet weak var loginButton: UIButton!
```

Добавим метод для настройки биндингов:

```
func configureLoginBindings() {
        Observable
// Объединяем два обсервера в один
            .combineLatest(
// Обсервер изменения текста
                loginView.rx.text,
// Обсервер изменения текста
                passwordView.rx.text
            )
// Модифицируем значения из двух обсерверов в один
            .map { login, password in
// Если введены логин и пароль больше 6 символов, будет возвращено "истина"
                return !(login ?? "").isEmpty && (password ?? "").count >= 6
            }
// Подписываемся на получение событий
            .bind { [weak loginButton] inputFilled in
// Если событие означает успех, активируем кнопку, иначе деактивируем
                loginButton?.isEnabled = inputFilled
        }
    }
```

Здесь **rx.text** генерирует событие каждый раз, когда в поле меняется текст. Но это два разных источника события. Мы соединяем их в один. Каждый раз, когда изменится значение одного из полей,

используется это новое и текущее значение второго поля, они передаются модификатору. Используемый модификатор преобразует два текстовых поля в одно логическое значение, которое мы передаём слушателю. Слушатель устанавливает полученное значение свойству активности кнопки.

Такой код можно написать и без **RxSwift**, но с ним он короче и выразительнее.

Теперь давайте отрефакторим контроллер карты. Настройка **CALocationManager** идёт прямо в нём, и это не очень хорошо. Вынесем это в отдельный класс.

```
import Foundation
import CoreLocation
import RxSwift

final class LocationManager: NSObject {
    static let instance = LocationManager()

    private override init() {
        super.init()
        configureLocationManager()
    }

    let locationManager = CLLocationManager()

    let location: Variable<CLLocation?> = Variable(nil)

    func startUpdatingLocation() {
        locationManager.startUpdatingLocation()
    }

    func stopUpdatingLocation() {
        locationManager.stopUpdatingLocation()
    }

    private func configureLocationManager() {
        locationManager.delegate = self
        locationManager.allowsBackgroundLocationUpdates = true
        locationManager.pausesLocationUpdatesAutomatically = false
        locationManager.desiredAccuracy = kCLLocationAccuracyNearestTenMeters
        locationManager.startMonitoringSignificantLocationChanges()
        locationManager.requestAlwaysAuthorization()
    }

}

extension LocationManager: CLLocationManagerDelegate {
        func locationManager(_ manager: CLLocationManager, didUpdateLocations
locations: [CLLocation]) {

        self.location.value = locations.last
    }

     func locationManager(_ manager: CLLocationManager, didFailWithError error:
Error) {
        print(error)
    }
}
```

Мы перенесли почти весь код работы с локацией в этот класс. Мы сделали его синглтоном: так он сможет быть использован в любом месте приложения, где потребуется текущее местоположение.

Ключевые в нём – две строки:

```
let location: Variable<CLLocation?> = Variable(nil)
```

```
self.location.value = locations.last
```

Первая строка – это объявление свойства типа **Variable**. Это обычная переменная, которая может хранить значение (правда, доступ к значению осуществляется через свойство **value)**. Это свойство – ещё и источник событий. Как только в переменную будет установлено новое значение, сразу же сгенерируется событие.

```
import UIKit
import GoogleMaps

class MapViewController: UIViewController {

    var usselesExampleVariable = ""

// Центр Москвы
       let coordinate = CLLocationCoordinate2D(latitude: 59.939095, longitude:
30.315868)
    let locationManager = LocationManager.instance
    var route: GMSPolyline?
    var routePath: GMSMutablePath?

    @IBOutlet weak var mapView: GMSMapView!

    override func viewDidLoad() {
        super.viewDidLoad()
        configureMap()
        configureLocationManager()
    }

    func configureMap() {
// Создаём камеру с использованием координат и уровнем увеличения
        let camera = GMSCameraPosition.camera(withTarget: coordinate, zoom: 17)
/ Устанавливаем камеру для карты
        mapView.camera = camera
    }

    func configureLocationManager() {
        locationManager
            .location
            .asObservable()
            .bind { [weak self] location in
                guard let location = location else { return }
                self?.routePath?.add(location.coordinate)
// Обновляем путь у линии маршрута путём повторного присвоения
                self?.route?.path = self?.routePath

// Чтобы наблюдать за движением, установим камеру на только что добавленную
// точку
                            let position = GMSCameraPosition.camera(withTarget:
```

```
location.coordinate, zoom: 17)
                self?.mapView.animate(to: position)
        }
    }

    @IBAction func updateLocation(_ sender: Any) {
// Отвязываем от карты старую линию
        route?.map = nil
// Заменяем старую линию новой
        route = GMSPolyline()
// Заменяем старый путь новым, пока пустым (без точек)
        routePath = GMSMutablePath()
// Добавляем новую линию на карту
        route?.map = mapView
// Запускаем отслеживание или продолжаем, если оно уже запущено
        locationManager.startUpdatingLocation()
    }
}
```

Это изменённый контроллер карты. Мы заменили **CALocationManager** на наш собственный **LocationManager** и вместо работы с делегатом подписались на изменение местоположения. Если нам одновременно с этим понадобится местоположение в другом месте, то мы и там точно так же сможем подписаться на изменение местоположения.

Теперь посмотрим, как создать собственный **Observable.** Например, создадим источник из запроса к серверу:

```
import Alamofire
import RxSwift

// Расширения, чтобы внедрить код сразу в вызов Alamofire
extension Request: ReactiveCompatible {}

// Расширения, чтобы внедрить код сразу в вызов Alamofire
extension Reactive where Base: DataRequest {

// Создаём свой аналог responseJSON, который будет возвращать Observable
    func responseJSON() -> Observable<Any> {
// Создаем Observable
        return Observable.create { observer in
// Используем реализацию из Alamofire, чтобы получить ответ
            let request = self.base.responseJSON { response in
// Обрабатываем успешный ответ
                switch response.result {
                case .success(let value):
// Публикуем событие с данными сервера
                    observer.onNext(value)
// Публикуем событие о завершении последовательности
                    observer.onCompleted()

                case .failure(let error):
// Если произошла ошибка, публикуем событие с ошибкой
                    observer.onError(error)
                }
            }
// Создаём инструкцию, как завершать Observable
            return Disposables.create(with: request.cancel)
        }
    }
```

```
}
```

Теперь мы сможем наблюдать и модифицировать запросы к серверу:

```
_ = Alamofire.request("https://httpbin.org/get").rx.responseJSON()
    .map { value in
        let json = value as? [String: Any] ?? [:]
        let origin = json["origin"] as? String ?? "unknown"
        return origin
    }
    .subscribe(onNext: { print("Origin:", $0) })
```

Чтобы прервать любое наблюдение источника, достаточно получить ссылку на подписку и вызвать у неё метод **dispose.**

```
let scheduler = SerialDispatchQueueScheduler(qos: .default)
let subscription = Observable<Int>.interval(0.3, scheduler: scheduler)
    .subscribe { event in
        print(event)
    }

Thread.sleep(forTimeInterval: 2.0)

subscription.dispose()
```

Это было бы отличным решением, но наблюдатели могут жить очень долго, теоретически бесконечно, и, как правило, их надо завершать вручную. Сохранять ссылки и вызывать у каждой из них **dispose** неудобно. Поэтому можно поступить по-другому: создать в классе свойство **let disposeBag = DisposeBag()** и передавать его каждому наблюдателю:

```
let scheduler = SerialDispatchQueueScheduler(qos: .default)
let subscription = Observable<Int>.interval(0.3, scheduler: scheduler)
    .subscribe { event in
        print(event)
    }
    .disposed(by: disposeBag)
```

Как только класс будет уничтожен, а с ним – и свойство **disposeBag,** все наблюдатели, которым оно было передано, будут остановлены.

Мы рассмотрели с вами основные приёмы работы с библиотекой RxSwift. В комплекте к ней идёт огромная масса операторов, которые позволяют модифицировать сигнал, но описывать их в методичке нет смысла: можно почитать официальную документацию или отличную статью на Хабре, где они все перечислены с примерами. Ссылки на них – в конце урока.

Практическое задание

1. Добавить в проект **RxSwift.**
2. Добавить наблюдателей на форму авторизации.
3. Блокировать кнопку «Войти», если поля не заполнены.
4. Вынести LocationManager в отдельный класс и организовать его наблюдение через **RxSwift.**

Дополнительные материалы

1. RxSwift шпаргалка по операторам.
2. RxSwift: работа с GUI.
3. Сайт библиотеки ReactiveX.
4. Репозиторий библиотеки ReactX.
5. RxSwift: Reactive Programming with Swift.

Используемая литература

1. RxSwift шпаргалка по операторам.
2. RxSwift: работа с GUI.
3. Сайт библиотеки ReactiveX.
4. Репозиторий библиотеки.