



# HOW TO DEAL WITH NAVIGATION

Andrei Rychkov

Всем привет! Сегодня, как вы догадались, мы поговорим о навигации в iOS приложениях. Речь пойдет о координаторах. Да, я знаю, что про них говорили уже года эдак 2 назад, но мне постоянно задают о них вопросы, поэтому я решил сделать этот доклад с описанием того, как я работаю с навигацией, а также затронуть несколько архитектурных моментов.

## ABOUT ME

- 5+ years in iOS development
- Worked on projects in various areas
- Now – Lead iOS Developer at FBS



2

Немного о себе – занимаюсь iOS разработкой почти 6 лет, работал над проектами в различных областях начиная с социальных сетей и заканчивая распознаванием лиц и шифрованием документов. Сейчас занимаюсь финансовыми приложениями в компании FBS.

## PLAN

1. Road trip
2. Meet coordinators!
3. FAQ

3

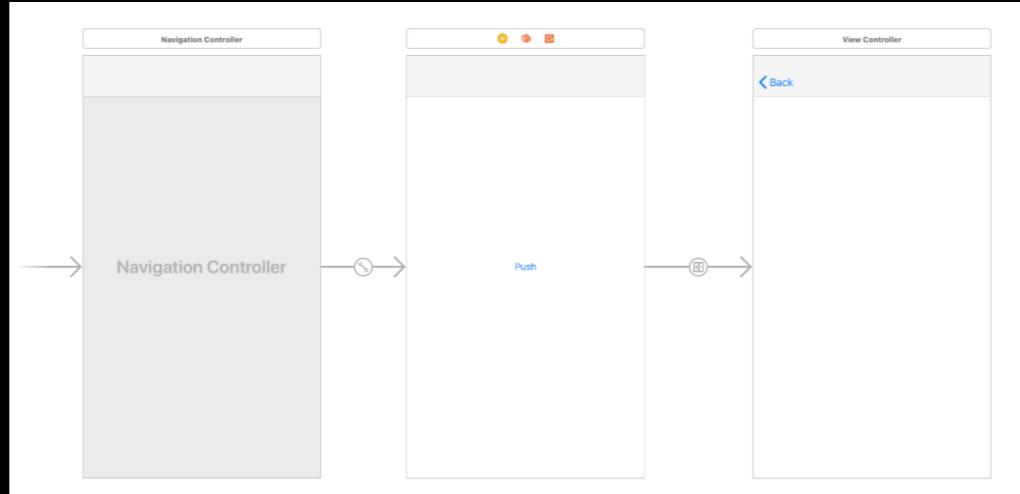
Сначала я расскажу, как обращался с навигацией раньше, почему отказался от тех или иных вариантов, а также предпосылки перехода к координаторам. Затем познакомлю с основными идеями этого подхода, а в конце постараюсь ответить на часто возникающие вопросы по реализации координаторов.

# ROAD TRIP



Итак, дорожное приключение. С чего же все началось мое знакомство с навигацией?

## STORYBOARD

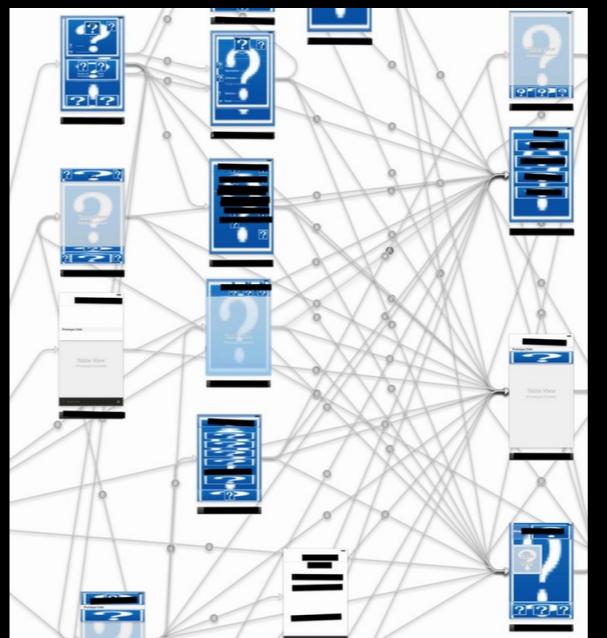


5

Ответ очевиден – вместе с iOS 5 появились сториборды и примерно в то же время я начал разработку на iOS. Все было хорошо, на небольших приложениях я видел все переходы между экранами, управлял ими с помощью интерфейса.

## THE PROBLEM

- Hard to maintain
- Hard to merge
- Strong relations between controllers



6

С течением времени мой сторибординг стал выглядеть примерно также, как на картинке. Его стало трудно поддерживать, мердж стал постоянной проблемой, а самое главное – между контроллерами устанавливались жесткие отношения, менять которые становилось все сложнее и сложнее. Разбиение на несколько сторибордингов, на первый взгляд, поможет, но это лишь уменьшит вероятность возникновения вышеуказанных проблем.

## PREPARE FOR SEGUE

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {  
    switch segue.destination {  
        case let dateController as DateViewController:  
            dateController.date = Date()  
        case let emailController as EmailViewController:  
            emailController.email = "cook@apple.com"  
        default:  
            break  
    }  
}
```

7

Метод `prepareForSegue()` достоин отдельного упоминания. Каждый раз, когда нам нужно передать данные между экранами, мы делаем typecast'ы, да еще и выставляем значение изменяемым переменным контроллеру напрямую.

## PRESENT / PUSH

```
func present(_ viewControllerToPresent: UIViewController,  
             animated flag: Bool,  
             completion: (() -> Swift.Void)? = nil)  
  
func pushViewController(_ viewController: UIViewController,  
                       animated: Bool)
```

8

Следующая остановка на моем пути – прямое использование двух методов показа контроллера – модальная презентация и добавление контроллера в навигейшн стек.

## TYPICAL TABLE VIEW

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    var controllerToPush: UIViewController
    let row = sections[indexPath.section].rows[indexPath.row]
    switch row {
        case let .emailDetails(email, server):
            controllerToPush = EmailDetailsViewController(email: email, server: server)
        case .passwordChange:
            controllerToPush = PasswordChangeViewController()
    }
    navigationController?.pushViewController(controllerToPush, animated: true)
}
```

9

Примерно так выглядит типичный метод делегата таблицы. Берем строчку, создаем следующий контроллер, говорим навигейшн контроллеру его запушить. Выделим основные проблемы.

## TYPICAL TABLE VIEW

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    var controllerToPush: UIViewController
    let row = sections[indexPath.section].rows[indexPath.row]
    switch row {
        case let .emailDetails(email, server):
            controllerToPush = EmailDetailsViewController(email: email, server: server)
        case .passwordChange:
            controllerToPush = PasswordChangeViewController()
    }
    navigationController?.pushViewController(controllerToPush, animated: true)
}
```

10

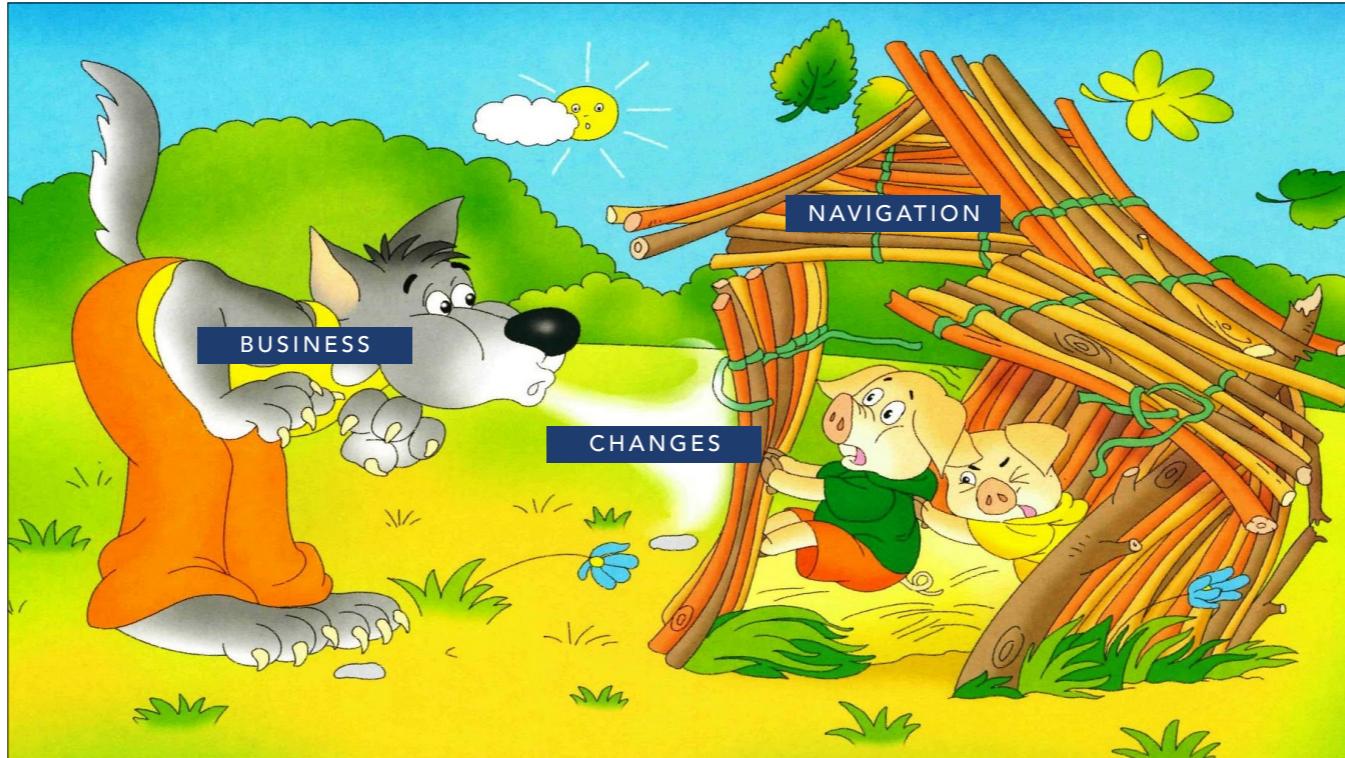
1 – Контроллер знает, как создать следующий контроллер и каким он будет. Да, сюда можно добавить фабрику, но, тем не менее, все варианты переходов будут заранее известны, добавление или изменение переходов влечет за собой изменение этого метода. И хорошо, если мы можем его изменить. Но контроллер может находиться, например, во фреймворке и быть неизменяемым.

## TYPICAL TABLE VIEW

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    var controllerToPush: UIViewController
    let row = sections[indexPath.section].rows[indexPath.row]
    switch row {
        case let .emailDetails(email, server):
            controllerToPush = EmailDetailsViewController(email: email, server: server)
        case .passwordChange:
            controllerToPush = PasswordChangeViewController()
    }
    navigationController?.pushViewController(controllerToPush, animated: true)
}
```

11

2 – Контроллер знает, что он находится в навигейшн стеке, при смене способа показа следующего контроллера опять же придется дописывать код этого.



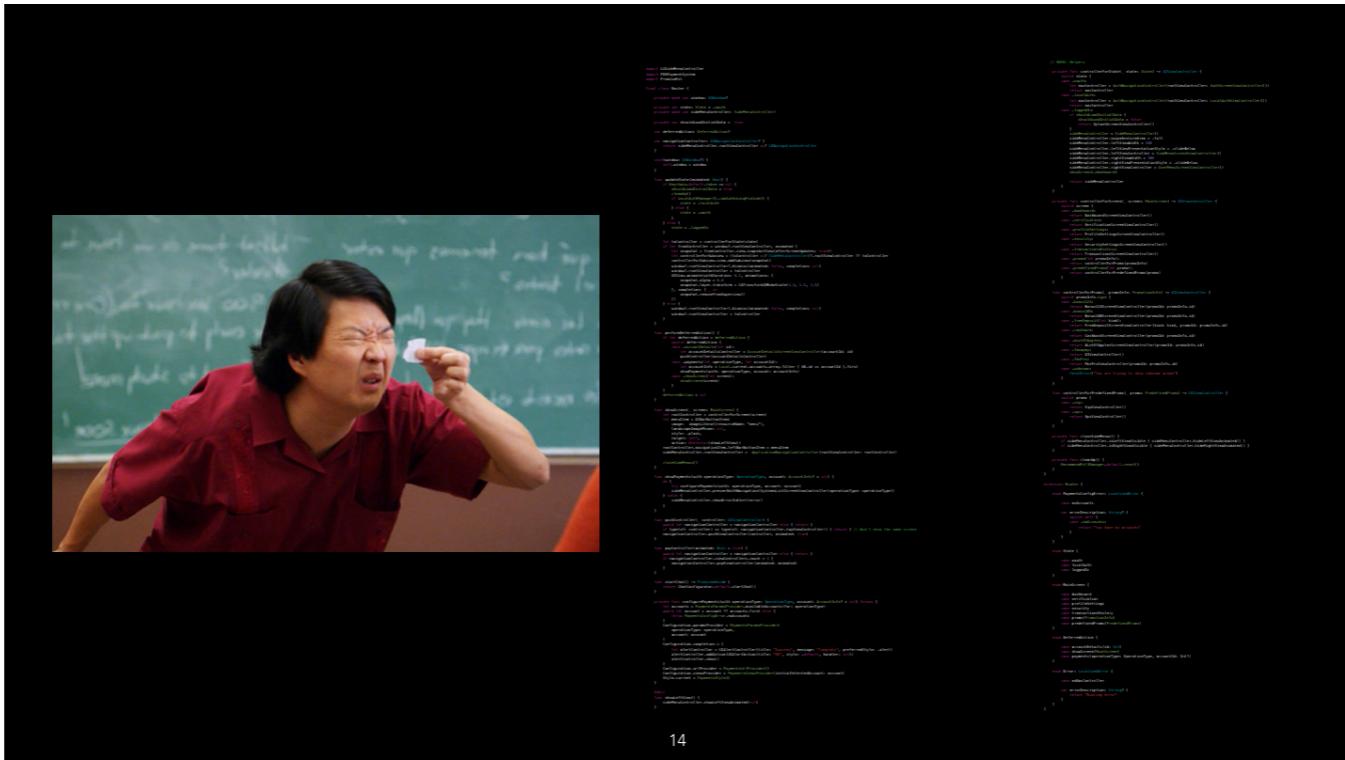
Как-то так выглядит наш код перед изменениями со стороны бизнеса

## ROUTER WILL CHANGE EVERYTHING

```
final class Router {  
  
    private weak var window: UIWindow?  
  
    private var state: State = .auth  
  
    private var navigationController: UINavigationController? {  
        return sideMenuController.rootViewController  
    }  
  
    func showPayments(with operationType: OperationType) {  
        let controller = SystemsListViewController(operationType: operationType)  
        navigationController?.pushViewController(controller, animated: true)  
    }  
}
```

13

Было бы круто иметь отдельный класс для навигации – подумал я и сделал Роутер. Не буду особо заострять внимание на деталях, скажу лишь в общих чертах. Этот класс управлял всей навигацией в проекте, имел несколько состояний, рулил дип линками и тому подобное. Сначала все было хорошо, я видел всю навигацию и легко в ней ориентировался, но есть одно но

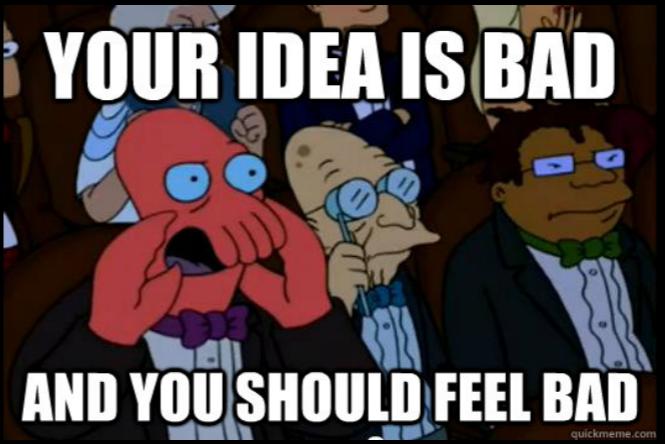


14

Через несколько месяцев в этом коде стало трудно ориентироваться, а связи между экранами были неочевидны. На экране видно часть реального кода роутера, я специально сделал помельче, чтобы показать масштаб трагедии.

## THE PROBLEM

- Router – GOD object
- It controls all screens navigation
- Has many different states



15

В общем, проблемы достаточно очевидны: получился GOD object, который знает обо всех перемещениях по приложению и имеет много состояний. Где-то в мире заплакала буква S из Солида

NEW PROJECT – NEW LIFE



16

Через некоторые времена у меня начался новый проект. Можно начать с чистого листа и сделать все хорошо.

## CONDITIONS

- Several applications
- Same flows
- Flows must be customisable
- Business logic often changes
- Can't easily change navigation in screen's code

17

Какие были условия: у компании будет несколько приложений, которые шарят одинаковые флоу (под флоу я имею ввиду логически связанные цепочки экранов типа флоу авторизации, профиля пользователя и так далее). Флоу могут немного отличаться в зависимости от приложения, что усложняет задачу. Но самое главное, это то, что бизнес логика может часто меняться. Из-за того, что приложений несколько, код делится на фреймворки, которые содержат реализацию фич, то есть реализация экранов закрыта от нас, и мы не можем просто так взять и поменять код любого контроллера, потребуется дописывать фреймворк.

## AUTH FLOW

18

Например, рассмотрим флоу авторизации.

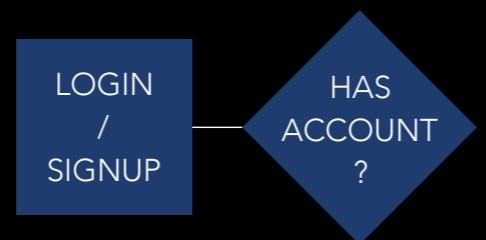
## AUTH FLOW

LOGIN  
/  
SIGNUP

19

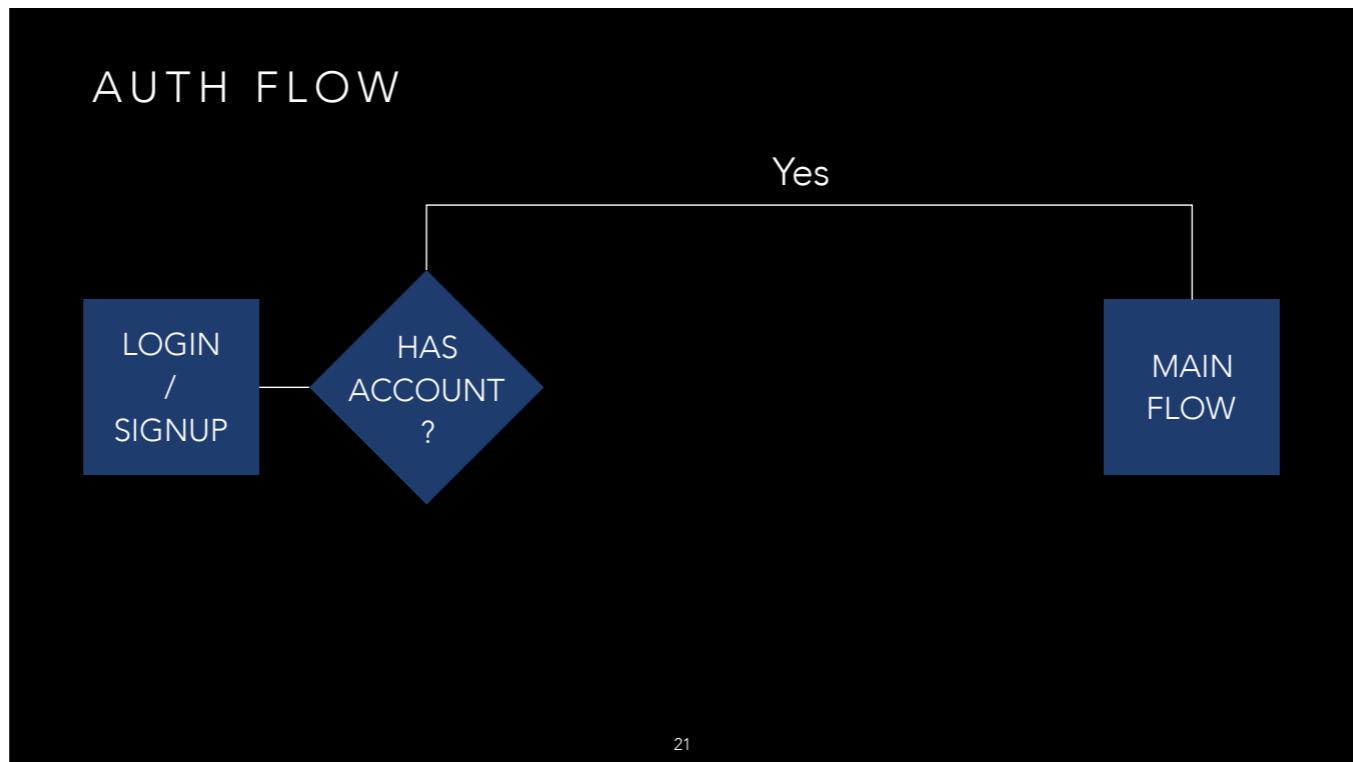
Сначала пользователь видит экраны логина или регистрации и авторизуется

## AUTH FLOW

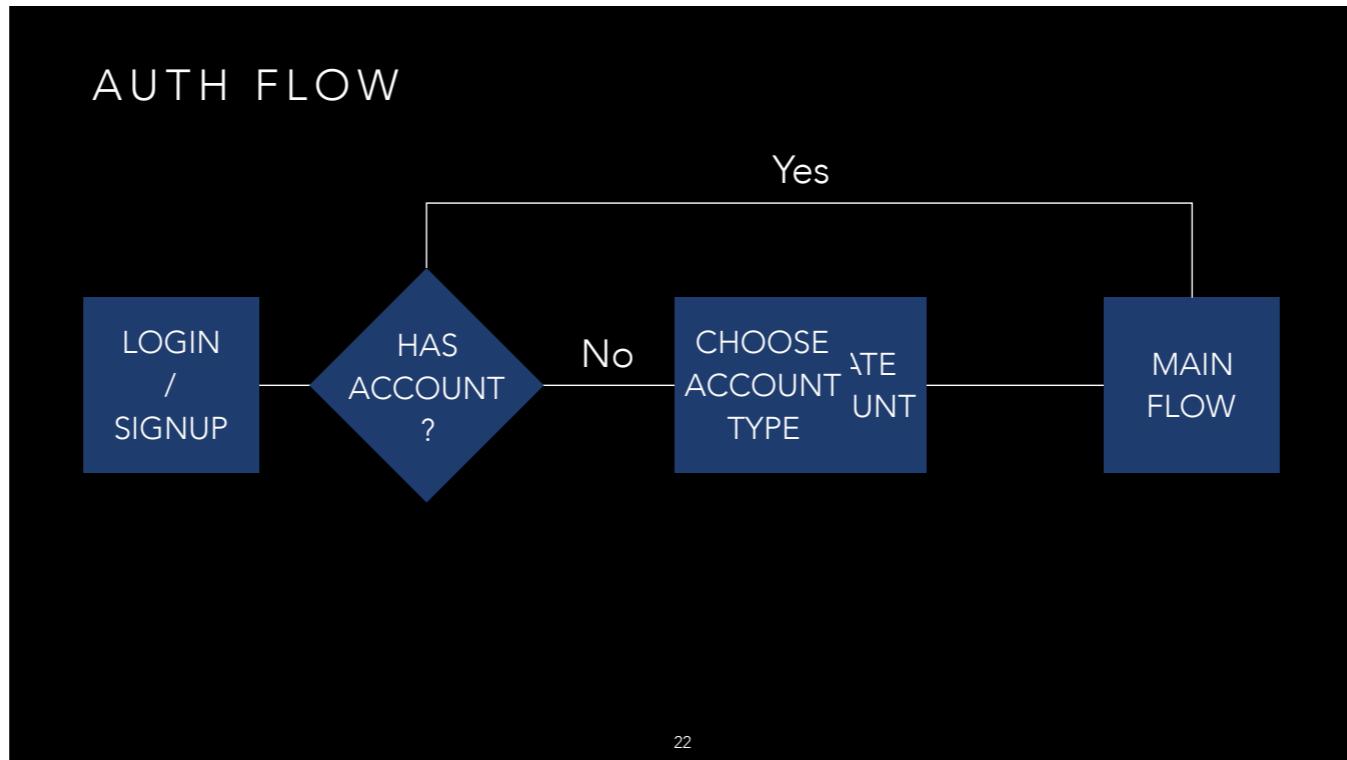


20

Затем необходимо проверить, если ли у него открытый банковский счет



Если есть, то отправляем клиента в основной флоу приложения



22

Если счета нет, то создадим его и пустим пользователя дальше. Вроде бы все просто. Но есть второе приложение, например, для других клиентов. В нем точно такой же флоу, но пользователь теперь может выбирать тип счета перед открытием. Получаем 2 одинаковых флоу, отличающиеся лишь одним экраном. А у третьего приложения может добавится / удалиться, скажем, еще один экран. Зачем остальным двум знать об этом варианте показа? В общем, я решил хорошо подготовиться к подобным случаям.

MEET COORDINATOR!



И координаторы оказались тем инструментом, который помог мне решить ранее перечисленные кейсы.

## THE IDEA



Andrey Panov [Follow](#)  
Lead iOS Developer in Blacklane  
Jan 12, 2017 · 8 min read

### Coordinators Essential tutorial. Part I



24

Основную идею я взял из статьи Андрея Панова, который на время её написания работал в Авито, а также у Соруша Ханлоу, я думаю, многие читали его блог. Ссылки будут на отдельном слайде в конце презентации. Сейчас же речь пойдет о том варианте, который я использую в своих проектах.

## MAIN POINTS

- Controller:
  - Doesn't directly present another controllers
  - Tells that its finished via delegate or closure
- Coordinator:
  - Listens to controller's completion
  - Decides which screen to show next
  - Shares flow control with child coordinators

25

Выделим основные пункты данного подхода – Теперь контроллер не показывает напрямую другие контроллеры, вместо этого он делегирует принятие решений наверх с помощью замыканий или делегата. Координатор в свою очередь является объектом, который управляет логикой навигации. Он слушает завершение у контроллера, решает, какой экран показать следующим, а также делится управлением флоу с дочерними координаторами. Например, флоу профиля пользователя может иметь дочерний флоу подтверждения телефона.

## PRESENTABLE

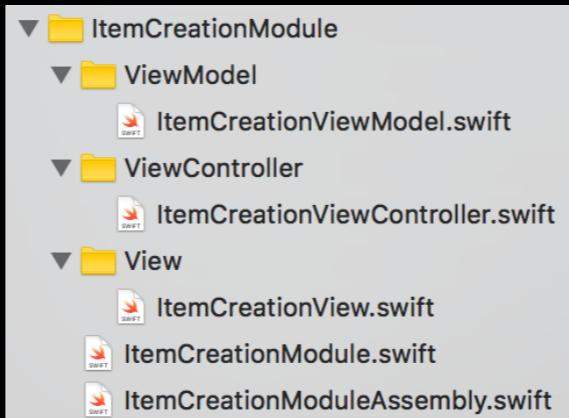
```
protocol Presentable {
    var toPresent: UIViewController? { get }
}

extension UIViewController: Presentable {
    var toPresent: UIViewController? {
        return self
    }
}
```

26

Как же это все выглядит? Начнем с протокола Presentable. Его требования – контроллер для презентации. По сути это обертка над объектом, который может предоставить VC для показа. Далеко не отходя, обернем в этот протокол виду контроллер.

## MODULE STRUCTURE



27

Все экраны делятся на модули – под модулем я имею ввиду одну архитектурную компоненту из MVC, VIPER или MVVM, которая предоставляет контроллер для презентации. В самом простом случае MVC это и есть контроллер. На слайде вы видите пример организации модуля с MVVM. Вью модель и вью ничем не отличаются от случая без координаторов, остальные же файлы рассмотрим чуть подробнее.

## MODULE

```
protocol ItemCreationModule: Presentable {

    typealias Completion = () -> Void

    var onFinish: Completion? { get set }
}

final class ItemCreationViewController: UIViewController,
    ItemCreationModule {

    var onFinish: Completion?
}
```

28

В протоколе модуля определяем блок завершения и объявляем соответствие протоколу в данном случае в контроллере.

ASSEMBLY



SWINJECT

Watch ▾ 75   Star 2,388   Fork 177

29

Отдельно упомяну работу с зависимостями. Для модульной организации контроля над зависимостями я использую библиотеку Swinject. В своем проекте вы можете заменить её на фабрики или любую другую библиотеку, но так как речь идет о моем опыте, дальше рассказ будет вестись в терминах Swinjecta, поэтому сейчас кратко поясню, как это работает.

```
protocol Animal {
    var name: String? { get }
}

class Cat: Animal {
    let name: String?

    init(name: String?) {
        self.name = name
    }
}
```

30

Есть протокол Animal, требующий у реализации кличку животного, а также реализация Cat с конструктором имени. Все просто.

```
let container = Container()
container.register(Animal.self) { resolver in
    return Cat(name: "Mimi")
}

let animal = container.resolve(Animal.self)!
print(animal.name) // "Mimi"
```

31

Создадим DI контейнер, в котором зарегистрируем наш протокол, передав фабрику в виде блока, который вернет нам кошку с именем Mimi. Теперь каждый раз, когда мы будем просить контейнер разрешить нам зависимость в виде протокола Animal, будет создаваться новый объект Cat с именем Mimi. В целом если не вдаваться в детали, то можно на всех следующих слайдах заменить блок регистрации на фабрики, а метод резолв на фабричный метод make

## ASSEMBLY

```
struct ItemCreationModuleAssembly: Assembly {

    func assemble(container: Container) {
        container.register(ItemCreationModule.self) { resolver in
            let controller = ItemCreationViewController()
            let viewModel = ItemCreationViewModelImpl()
            viewModel.itemsService = resolver.resolve(ItemsService.self)
            controller.viewModel = viewModel
            return controller
        }
    }
}
```

32

Как вы видели на слайде со структурой проекта, у каждого модуля есть своя сборка. Это объект, который предоставляет контейнер для регистрации там нужных объектов. В блоке фабрики создаются все наши архитектурные компоненты и проставляются нужные зависимости, которые мы можем просить разрешить для нас путем вызова метода resolve()

## COORDINATOR

```
protocol Coordinator: class {  
  
    var router: Routable { get }  
  
    func start()  
    func start(with option: DeepLinkOption?)  
}
```

33

Сам координатор, напомню, отвечает за управление флоу, то есть выбирает, какой экран показать следующим, и просит роутер это сделать. Он должен иметь объект, называемый роутером, для непосредственно показа модулей, а также метод старта флоу с опциональным параметром динлинка.

## ROUTER

```
protocol Routable: Presentable {

    func setRootModule(_ module: Presentable?, animated: Bool)
    func push(_ module: Presentable?, animated: Bool)
    func popModule(animated: Bool)

    func present(_ module: Presentable?, animated: Bool)
    func dismissModule(animated: Bool, completion: (() -> Void)?)
}
```

34

Routable отвечает за осуществление переходов между экранами. Таким образом, координатор ничего не знает о реализации показа модулей. Чаще всего вы будете использовать роутер как обертку над навигейшн контроллером, где 3 первых метода работают со стеком контроллеров, а 2 последних показывают и скрывают модальные контроллеры. Но бывают ситуации, когда роутер может пригодиться. Например, дизайнеры решили сделать навигацию по поп-апам. Скрывается один, показывается другой. Путем замены роутера можно легко заменить способ навигации, а когда дизайнеры опомнятся, щелчком пальцев вернуть обратно.

## DEEP LINKING

```
protocol DeepLinkOption {  
  
    static func build(with userActivity: NSUserActivity) -> DeepLinkOption?  
    static func build(with dict: [String: AnyObject]?) -> DeepLinkOption?  
}
```

35

Что касается DeepLinkOption, который, как следует из названия, оборачивает диплинки , то он требует от объекта конструкторы с перехода по URL или нотификации.

## BASE COORDINATOR

```
open class BaseCoordinator: Coordinator {

    var childCoordinators: [Coordinator] = []
    let router: Routable
    let assembler: Assembler

    init(assembler: Assembler, router: Routable) {
        self.assembler = assembler
        self.router = router
    }

    open func start(with option: DeepLinkOption?) { }
}
```

36

Для удобства введем базовый класс координатора, у которого есть массив дочерних координаторов для удобства удерживания, роутер и ассемблер. Метод start() оставляем пустым. Ассемблер в данном случае – сборщик зависимостей, из которого мы их и будем доставать.

```
extension BaseCoordinator {

    func addDependency(_ coordinator: Coordinator) {
        guard !childCoordinators.contains(where: { $0 === coordinator })
            else { return }
        childCoordinators.append(coordinator)
    }

    func removeDependency(_ coordinator: Coordinator?) {
        guard let indexToRemove = childCoordinators.index(where: { $0 === coordinator })
            else { return }
        childCoordinators.remove(at: indexToRemove)
    }

    func removeAllDependencies() {
        childCoordinators.removeAll()
    }
}
```

37

Также определим методы работы с дочерними координаторами. Ничего особенного, просто перед добавлением или удалением элемента проверяем его наличие в массиве. Не будем здесь задерживаться и перейдем к примерам.

## INIT

```
class AppDelegate: UIResponder, UIApplicationDelegate {  
    var window: UIWindow?  
    private lazy var appCoordinator: Coordinator = makeAppCoordinator()  
    func application(... didFinishLaunchingWithOptions...) -> Bool {  
        window = UIWindow(frame: UIScreen.main.bounds)  
        let notification = launchOptions?[.remoteNotification] as? [String: AnyObject]  
        let deepLink = CEDeepLinkOption.build(with: notification)  
        appCoordinator.start(with: deepLink)  
        window?.makeKeyAndVisible()  
        return true  
    }  
}
```

Для начала посмотрим на инициализацию графа зависимостей и навигации.

## INIT

```
class AppDelegate: UIResponder, UIApplicationDelegate {  
    var window: UIWindow?  
  
    private lazy var appCoordinator: Coordinator = makeAppCoordinator()  
  
    func application(... didFinishLaunchingWithOptions...) -> Bool {  
  
        window = UIWindow(frame: UIScreen.main.bounds)  
        let notification = launchOptions?[.remoteNotification] as? [String: AnyObject]  
        let deepLink = CEDeepLinkOption.build(with: notification)  
        appCoordinator.start(with: deepLink)  
        window?.makeKeyAndVisible()  
  
        return true  
    }  
}
```

39

Главный координатор приложения создается при запуске и удерживается сильной ссылкой.

## INIT

```
class AppDelegate: UIResponder, UIApplicationDelegate {  
    var window: UIWindow?  
    private lazy var appCoordinator: Coordinator = makeAppCoordinator()  
  
    func application(...didFinishLaunchingWithOptions...) -> Bool {  
        window = UIWindow(frame: UIScreen.main.bounds)  
        let notification = launchOptions?[.remoteNotification] as? [String: AnyObject]  
        let deepLink = CEDeepLinkOption.build(with: notification)  
        appCoordinator.start(with: deepLink)  
        window?.makeKeyAndVisible()  
  
        return true  
    }  
}
```

40

В методе didFinishLaunching проверяем, был ли это переход по нотификации или нет, а затем запускаем координатор приложения.

```
extension AppDelegate {

    func makeAppCoordinator() -> Coordinator {
        let rootAssembler = Assembler(
            [
                ServicesAssembly(),
                AppCoordinatorAssembly()
            ],
            container: Container()
        )
        return rootAssembler.resolver.resolve(
            AppCoordinator.self,
            arguments: rootAssembler, window
        )!
    }
}
```

41

Посмотрим поближе на makeAppCoordinator().

```
extension AppDelegate {

    func makeAppCoordinator() -> Coordinator {
        let rootAssembler = Assembler(
            [
                ServicesAssembly(),
                AppCoordinatorAssembly()
            ],
            container: Container()
        )
        return rootAssembler.resolver.resolve(
            AppCoordinator.self,
            arguments: rootAssembler, window
        )!
    }
}
```

42

Создаем главный сборщик

```
extension AppDelegate {

    func makeAppCoordinator() -> Coordinator {
        let rootAssembler = Assembler(
            [
                ServicesAssembly(),
                AppCoordinatorAssembly()
            ],
            container: Container()
        )
        return rootAssembler.resolver.resolve(
            AppCoordinator.self,
            arguments: rootAssembler, window
        )!
    }
}
```

43

В который помещаем сборку сервисного слоя приложения

```
extension AppDelegate {

    func makeAppCoordinator() -> Coordinator {
        let rootAssembler = Assembler(
            [
                ServicesAssembly(),
                AppCoordinatorAssembly()
            ],
            container: Container()
        )
        return rootAssembler.resolver.resolve(
            AppCoordinator.self,
            arguments: rootAssembler, window
        )!
    }
}
```

44

И сборку координатора приложения

```
extension AppDelegate {

    func makeAppCoordinator() -> Coordinator {
        let rootAssembler = Assembler(
            [
                ServicesAssembly(),
                AppCoordinatorAssembly()
            ],
            container: Container()
        )
        return rootAssembler.resolver.resolve(
            AppCoordinator.self,
            arguments: rootAssembler, window
        )!
    }
}
```

45

А также контейнер для регистрации объектов

```
extension AppDelegate {

    func makeAppCoordinator() -> Coordinator {
        let rootAssembler = Assembler(
            [
                ServicesAssembly(),
                AppCoordinatorAssembly()
            ],
            container: Container()
        )
        return rootAssembler.resolver.resolve(
            AppCoordinator.self,
            arguments: rootAssembler, window
        )!
    }
}
```

46

И, наконец, резолвим и возвращаем координатор приложения, передав сборке необходимые аргументы – ассемблер и UIWindow, которое будет использовано для создания роутера приложения.

```
final class AppCoordinatorImpl: BaseCoordinator, AppCoordinator {

    override func start(with option: DeepLinkOption?) {
        if authenticated {
            runMainFlow(with: option)
        } else {
            runAuthFlow(with: option)
        }
    }

    private func runAuthFlow(with deepLink: DeepLinkOption? = nil) {
        let coordinator = assembler.resolver.resolve(AuthCoordinator.self, argument: assembler)!
        coordinator.onFinish = { [weak self] deepLink in
            self?.removeDependency(coordinator)
            self?.runMainFlow(with: deepLink)
        }
        addDependency(coordinator)
        router.setRootModule(coordinator.router)
        coordinator.start(with: deepLink)
    }

    private func runMainFlow(with deepLink: DeepLinkOption?) { . . . }
}
```

47

Посмотрим на реализацию.

```
final class AppCoordinatorImpl: BaseCoordinator, AppCoordinator {

    override func start(with option: DeepLinkOption?) {
        if authenticated {
            runMainFlow(with: option)
        } else {
            runAuthFlow(with: option)
        }
    }

    private func runAuthFlow(with deepLink: DeepLinkOption? = nil) {
        let coordinator = assembler.resolver.resolve(AuthCoordinator.self, argument: assembler)!
        coordinator.onFinish = { [weak self] deepLink in
            self?.removeDependency(coordinator)
            self?.runMainFlow(with: deepLink)
        }
        addDependency(coordinator)
        router.setRootModule(coordinator.router)
        coordinator.start(with: deepLink)
    }

    private func runMainFlow(with deepLink: DeepLinkOption?) { . . . }
}
```

48

В методе старта координатор решает, какой флоу запустить, авторизации или главный флоу приложения

```
final class AppCoordinatorImpl: BaseCoordinator, AppCoordinator {

    override func start(with option: DeepLinkOption?) {
        if authenticated {
            runMainFlow(with: option)
        } else {
            runAuthFlow(with: option)
        }
    }

    private func runAuthFlow(with deepLink: DeepLinkOption? = nil) {
        let coordinator = assembler.resolver.resolve(AuthCoordinator.self, argument: assembler)!
        coordinator.onFinish = { [weak self] deepLink in
            self?.removeDependency(coordinator)
            self?.runMainFlow(with: deepLink)
        }
        addDependency(coordinator)
        router.setRootModule(coordinator.router)
        coordinator.start(with: deepLink)
    }

    private func runMainFlow(with deepLink: DeepLinkOption?) { . . . }
}
```

49

А в методе запуска флоу авторизации просим у сборщика реализацию координатора, передавая туда необходимые аргументы, подписываемся на окончание работы дочернего координатора, удерживаем его в списке зависимостей, говорим роутеру сделать рутовым презентером роутер созданного координатора, и запускаем его с потенциальным дип линком. В блоке подписки собственно убираем зависимость и запускаем главный флоу.

```
final class MainCoordinatorImpl: BaseCoordinator, MainCoordinator {  
    . . .  
    private func showItemsList() {  
        var module = assembler.resolver.resolve(ItemsListModule.self)  
        module?.onItemCreate = { [weak self] in  
            self?.runItemCreationFlow(animated: true)  
        }  
        module?.onItemSelected = showDetails  
        module?.onLogout = onLogout  
        router.push(module, animated: false)  
    }  
    . . .  
}
```

Показ конкретных модулей практически ничем не отличается от запуска флоу. Все также резолвим реализацию модуля, подписываемся на различные коллбэки и просим роутер показать нужный модуль.

```
struct MainCoordinatorAssembly: Assembly {

    func assemble(container: Container) {
        container.register(MainCoordinator.self) { (resolver, parentAssembler: Assembler) in
            let assembler = Assembler(
                [
                    ItemsListModuleAssembly(),
                    ItemDetailsModuleAssembly(),
                    ItemCreationCoordinatorAssembly()
                ],
                parent: parentAssembler
            )
            let router = NavigationRouter(rootController: UINavigationController())
            let coordinator = MainCoordinatorImpl(assembler: assembler, router: router)
            return coordinator
        }
    }
}
```

51

Сама типичная сборка координатора выглядит вот так:

```
struct MainCoordinatorAssembly: Assembly {

    func assemble(container: Container) {
        container.register(MainCoordinator.self) { (resolver, parentAssembler: Assembler) in
            let assembler = Assembler(
                [
                    ItemsListModuleAssembly(),
                    ItemDetailsModuleAssembly(),
                    ItemCreationCoordinatorAssembly()
                ],
                parent: parentAssembler
            )
            let router = NavigationRouter(rootController: UINavigationController())
            let coordinator = MainCoordinatorImpl(assembler: assembler, router: router)
            return coordinator
        }
    }
}
```

52

Регистрируем в контейнере протокол MainCoordinator с необходимым параметром сборщика

```
struct MainCoordinatorAssembly: Assembly {

    func assemble(container: Container) {
        container.register(MainCoordinator.self) { (resolver, parentAssembler: Assembler) in
            let assembler = Assembler(
                [
                    ItemsListModuleAssembly(),
                    ItemDetailsModuleAssembly(),
                    ItemCreationCoordinatorAssembly()
                ],
                parent: parentAssembler
            )
            let router = NavigationRouter(rootController: UINavigationController())
            let coordinator = MainCoordinatorImpl(assembler: assembler, router: router)
            return coordinator
        }
    }
}
```

53

Для этого создаем дочерний сборщик, передавая ему необходимые сборки модулей и координаторов

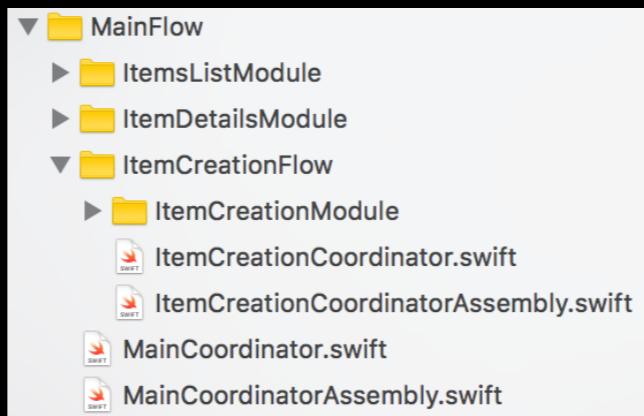
```
struct MainCoordinatorAssembly: Assembly {

    func assemble(container: Container) {
        container.register(MainCoordinator.self) { (resolver, parentAssembler: Assembler) in
            let assembler = Assembler(
                [
                    ItemsListModuleAssembly(),
                    ItemDetailsModuleAssembly(),
                    ItemCreationCoordinatorAssembly()
                ],
                parent: parentAssembler
            )
            let router = NavigationRouter(rootController: UINavigationController())
            let coordinator = MainCoordinatorImpl(assembler: assembler, router: router)
            return coordinator
        }
    }
}
```

54

Затем создаем нужный роутер и возвращаем координатор

## FLOW STRUCTURE



55

По структуре проекта – В папке флоу лежат относящиеся к нему модули, дочерние флоу, а также сам координатор вместе со своей сборкой.

## FAQ



По основным идеям все, теперь я постараюсь ответить на несколько часто задаваемых вопросов по поводу координаторов

## HOW TO CONFIGURE MODULES IN NAVIGATION?

```
extension Presentable {

    func withRemovedBackItem() -> Presentable? {
        toPresent?.navigationItem.hidesBackButton = true
        return self
    }

    func withHiddenBottomBar() -> Presentable? {
        toPresent?.hidesBottomBarWhenPushed = true
        return self
    }
}
```

57

Некоторые экраны могут иметь различные конфигурации, находясь в навигейшн стеке. Конфигурацию можно вынести из самих модулей, сделав расширение, которое позволяет их конфигурировать в координаторе, например, спрятать кнопку назад или таб бар. Особенно мне это пригодилось на флоу регистрации. Когда пользователь может в середине флоу закрыть приложение и зайти обратно, необходимо вернуть его на то же место, где он и закончил. Из-за этого конфигурация одного и того же экрана может отличаться. В репозитории с примером (ссылка будет в конце презентации) вы найдете еще несколько хелперов.

## WHAT ABOUT DEEP LINKING?

```
private func runAuthFlow(with deepLink: DeepLinkOption? = nil) {  
    let coordinator = [AuthCoordinator creation]  
    coordinator.onFinish = { in  
        [...]  
        runMainFlow(with: deepLink)  
    }  
    [...]  
    coordinator.start(with: deepLink)  
}
```

58

С дип линками все довольно просто. Они создаются в соответствующих методах апп делегата, как мы видели ранее, а затем передаются в метод start() координатору. Например, дин линк, который требует авторизации пользователя, можно в несколько строк прокинуть через весь флоу авторизации (который может быть немаленьким) и открыть какой-нибудь экран в главном флоу. Также никто не запрещает посмотреть, запущен ли какой-то дочерний флоу и сделать действие на нем не пересоздавая весь стэк.

## BACK BUTTON BREAKS THE GAME?

- We call dismiss directly
- Back button and interactive pop work automatically
- We don't know when to dispose the coordinator
- Make navigation router implement UINavigationControllerDelegate
- Catch popped view controller and call its completion

59

Кто бы мог подумать, но кнопка назад в навигешн баре и интерактивный жест могут стать проблемой. Мы сами вызываем закрытие модального окна, а кнопка назад или интерактивный жест в UINavigationController'е работают автоматически. Получается, что мы не знаем, когда экран закрывается, и не можем избавится от координатора. Есть несколько решений, которые можно найти в интернете, больше всего мне нравится такое: сделать роутер делегатом навигейшн контроллера, ловить закрытый экран и вызывать его завершающий блок.

## HOW TO MAKE INITIAL DECISION?

```
final class LaunchInstructor {

    enum Instruction {
        case onboarding
        case auth
        case main
    }

    var instruction: Instruction {
        guard passedOnboarding else { return .onboarding }
        switch authStateProvider.authState {
        case false:
            return .auth
        case true:
            return .main
        }
    }
}
```

60

Для определения начального состояния флоу, например, выбора, какой экран показать при запуске, можно использовать дополнительные объекты, которые предоставляют инструкции по тем или иным вариантам. Чтобы принять решение эти инструкторы могут обращаться за состояниями в другие объекты, на слайде, например, в провайдер состояния авторизации.

## HOW TO PASS THE DATA?

- Just pass data through methods
- Store flow data in coordinator via dictionary
- Store flow state in coordinator via storage object

61

Есть несколько вариантов передачи данных между флоу и модулями во флоу. Первый – не заморачиваться и просто передавать данные с выхода одного модуля на вход другому, из-за чего мы теряем в гибкости. Второй – хранить данные в координаторе с помощью словаря по ключам в виде епим'ов, и последний – сохранять состояние флоу в специально созданном для этого объекте, получая его для следующего экрана.

## FLOW CONTROLLER? SOUNDS THE SAME!

- Flow controller is an UIViewController subclass
- Flow controller adds dependencies as child view controllers
- Coordinator is separated from UIKit == more flexible

62

Есть аналог координатору – флоу контроллер. В целом, идея та же, но флоу контроллер наследуется от UIViewController'а и работает напрямую с UIKit'ом, добавляя зависимости как childViewController. В отличие от координатора, который в свою очередь отрезан от UIKit'а набором абстракций. Если честно, то особо не понимаю, зачем сильно привязываться к UIKit'у, разве что для сохранения времени на описание абстракций или более точечного управления флоу. Есть отдельная статья с описанием преимуществ флоу контроллера над координаторами, ссылка также появится в конце доклада.

## WORKS WITH VIPER?!

- Navigation logic is inside VIPER module
- Maybe need to remove Router and listen to events from Presenter
- VIPEC

63

С вайпером все довольно сложно, потому что логика навигации уже зашита внутрь VIPER модуля. Он знает, какой контроллер нужно показать дальше. Чтобы добавить координатор, надо убрать роутер и слушать события с презентера, то есть получается VIPEC и это наверняка стоит темы отдельного доклада :) Если по этим вопросам будут какие-то идеи –подходите после доклада, обсудим.

## OK, I'M READY! WHAT'S NEXT?

- Separate your navigation logic into flows
- Start refactoring them one by one starting with root
- Suffer
- Live happily

64

Допустим, вы решились использовать координаторы. Если у вас новое приложение, то говорить особо не о чем – с самого начала используете координаторы, и будет вам счастье. Если же у вас старый проект, то разбивать придется по частям. Сначала сядьте и разделите все ваши перемещения на флоу. Затем начните рефакторить ваши флоу один за другим. Страдайте, потому что потребуется добавить определенное количество костылей. Например, когда я мигрировал относительно небольшой проект на координаторы, приходилось отправлять некоторые события через Notification Center. И, наконец, живите счастливо с координаторами :)

## CONCLUSION

- We created reusable components – flows and modules
- Modules don't know about each other
- They just do they work – receive input and return output
- Now we can easily customise our flows

65

Подведя итоги можно сказать, что мы получили переиспользуемые компоненты – флоу и модули. Модули ничего не знают о других модулях, что опять же упрощает переиспользование. Они делают то, для чего и задумывались – принимают данные, обрабатывают их и отдают что-то на выход. Теперь мы можем легко кастомизировать наши флоу, не трогая код самих экранов.

<https://github.com/Moonko/CoordinatorExample>



66

За этим QR кодом находится ссылка на пример реализации координаторов. Давайте остановимся тут ненадолго, чтобы все желающие могли сфотографировать.

## USEFUL LINKS

- [Andrei Panov – Coordinators Essential](#)
- [Soroush Khanlou – Presenting coordinator](#)
- [Flow controller iOS](#)
- [Onmyway133 – Coordinator and Flow controller](#)
- [Swinject](#)

67

На этом слайде вы найдете полезные ссылки по теме координаторов. Презентацию должны выложить в открытый доступ, вы сможете их посмотреть. Также можете сделать фото, все ссылки будут первыми в гугле, если поискать по одной из этих строчек.

## THANKS FOR YOUR ATTENTION

```
private func runMainFlow(with deepLink: DeepLinkOption?) {
    let coordinator = coordinatorsFabric.makeMainCoordinator()
    coordinator.onLogout = { [weak self] in
        self?.removeDependency(coordinator)
        self?.runAuthFlow()
    }
    addDependency(coordinator)
    router.setRootModule(coordinator.router)
    coordinator.start(with: deepLink)
}
```

ANY ERRORS?

a\_rychkov 

68

Справа в углу мой аккаунт в заблокированном мессенджере, туда тоже можно задавать вопросы. Также небольшой тест на внимательность – найдите здесь довольно серьезную ошибку, которую легко может допустить каждый. С ответом подходите после доклада, или его можно подсмотреть в моем репозитории. Спасибо за внимание!