iOS Meetup Munich

XCoordinator

A powerful navigation library for iOS

What we will build

We will refactor a MVC app to encapsulate transition logic and manage inter-scene dependencies with XCoordinator.

What you will learn

What the Coordinator pattern is
How to make inter-scene dependencies clear
How XCoordinator makes code transitions easier
How to use animations in XCoordinator
Deep links, Reactive extensions & more

Key Vocabulary

- Coordinator pattern
- XCoordinator
- Transitions
- Animations
- Dependencies
- Swift

Copyright, Content & Referrals and Links

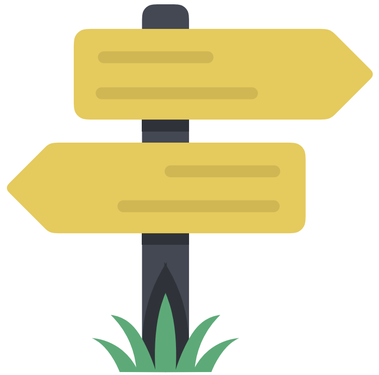
- **Copyright 2019 QuickBird Studios**
  - Unless explicitly stated otherwise, all rights including those in copyright in the content of this document are owned by or controlled for these purposes by QuickBird Studios.
  - Except as otherwise expressly permitted under copyright law or QuickBird Studios's Terms of Use, the content of this document may not be copied, reproduced, republished, posted, broadcast or transmitted in any way without first obtaining QuickBird Studios's written permission.
- **Content**
  - QuickBird Studios reserves the right not to be responsible for the topicality, correctness, completeness or quality of the information provided.
  - Liability claims regarding damage caused by the use of any information provided, including any kind of information which is incomplete or incorrect, will therefore be rejected.
  - All offers are not-binding and without obligation. Parts of the pages or the complete publication including all offers and information might be extended, changed or partly or completely deleted by the author without separate announcement.
- **Referrals and Links**
  - The author is not responsible for any contents linked or referred to from this document.
  - If any damage occurs by the use of information presented there, only the author of the respective pages might be liable, not the one who has linked to these pages.
  - Furthermore the author is not liable for any postings or messages published by users of discussion boards, guestbooks or mailing lists provided on his page.
- **By obtaining and reading this document, you agree to this copyright statement.**

QuickBird Studios

Agenda

Common practice

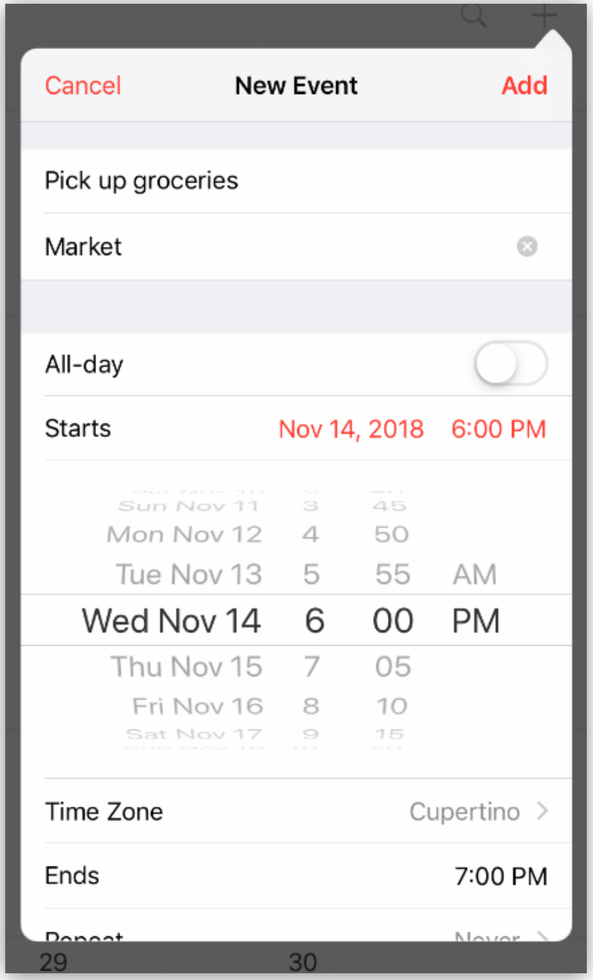
Coordinator pattern

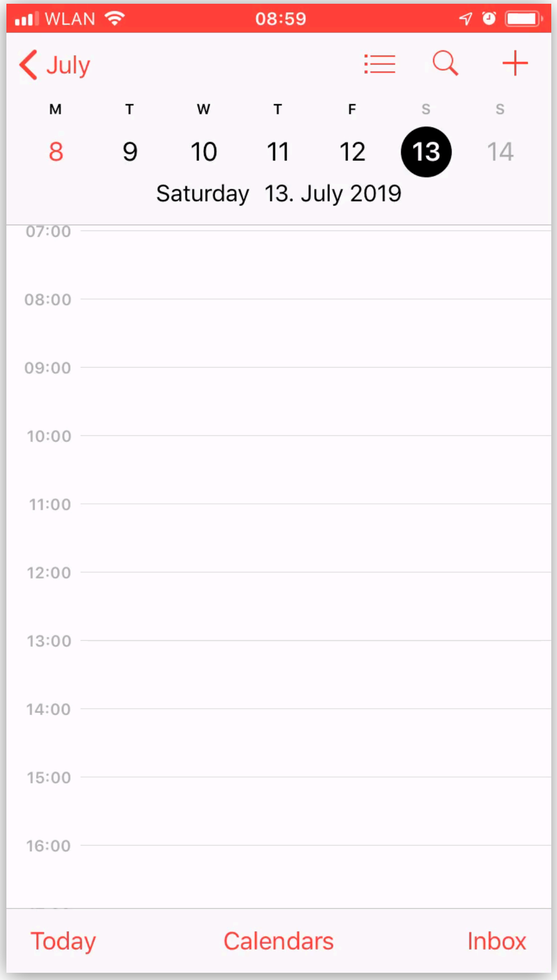


XCoordinator

Live coding

What are transitions?

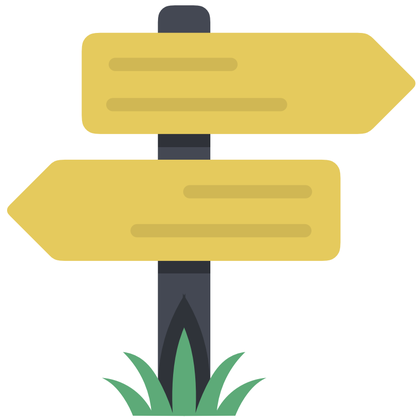




Navigation techniques

Storyboard Segues

UIKit transition code



XCoordinator

Storyboard Segues

```
override func prepare(for segue: UIStoryboardSegue,
                      sender: Any?) {
    switch segue.identifier {
    case "PushNewViewController":
        guard let viewController =
            segue.destination as? NewViewController else {
            return assertionFailure()
        }
        viewController.data = Data()
    default:
        assertionFailure()
    }
}
```

String constants

assumptions about viewController context

viewController-dependent transitions

no type-safety for transition destination

Transitions in code

```
@IBAction func loginButtonTapped(_ sender: UIButton) {
    guard let navigationController = navigationController else {
        return assertionFailure()
    }

    let viewController = NewViewController()
    viewController.data = Data()
    navigationController.pushViewController(viewController, animated: true)
}
```

no String constants

assumptions about viewController context

viewController-dependent transitions

type-safety for transition destination

Transitions using a Coordinator pattern

LoginViewController:

```
@IBAction func loginButtonTapped(_ sender: UIButton) {
    coordinator.trigger(.loginSuccessful)
}
```

LoginCoordinator:

```
override func prepareTransition(for route: LoginRoute) -> NavigationTransition {
    switch route {
    case .loginSuccessful:
        let viewController = NewViewController()
        viewController.data = Data()
        return .push(viewController)
    }
}
```

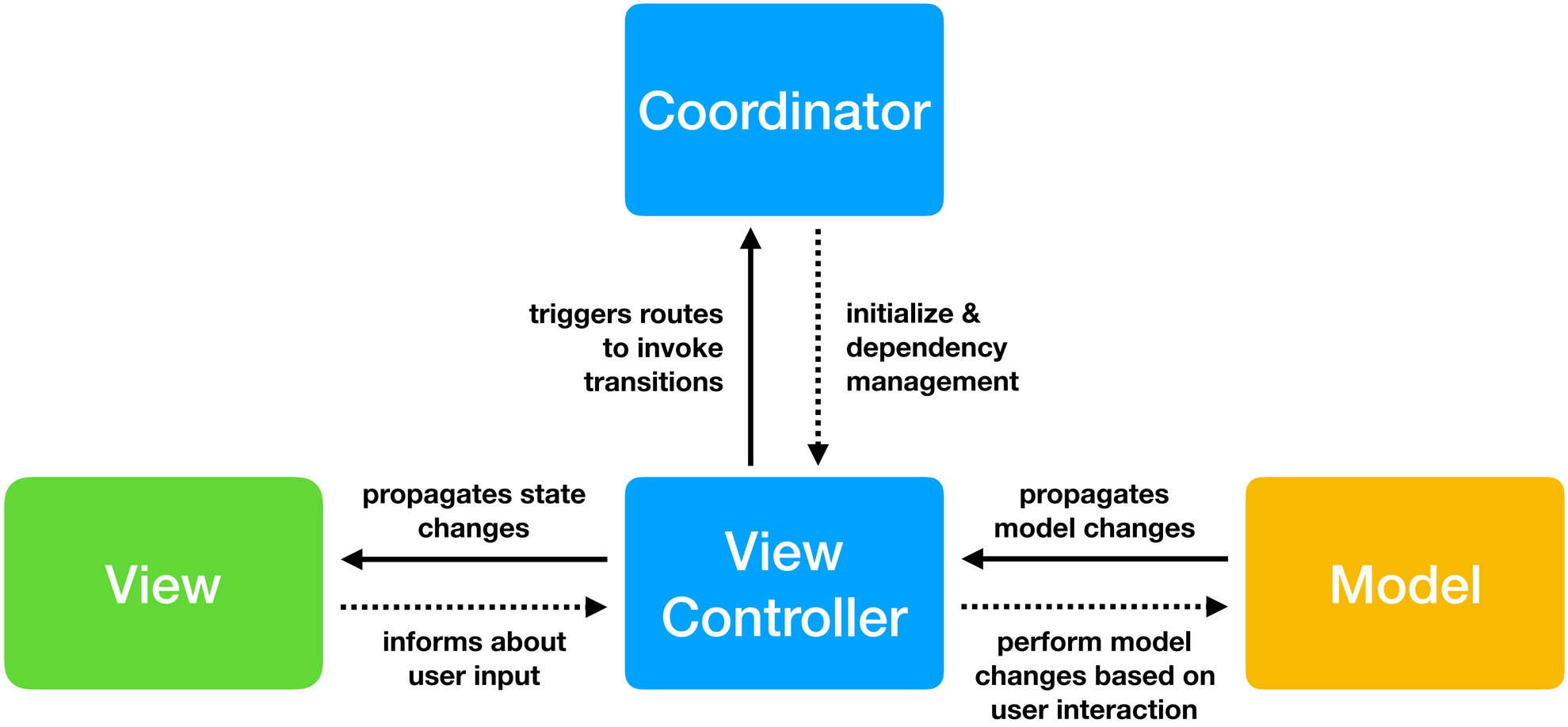
no String constants

no assumptions about viewController context

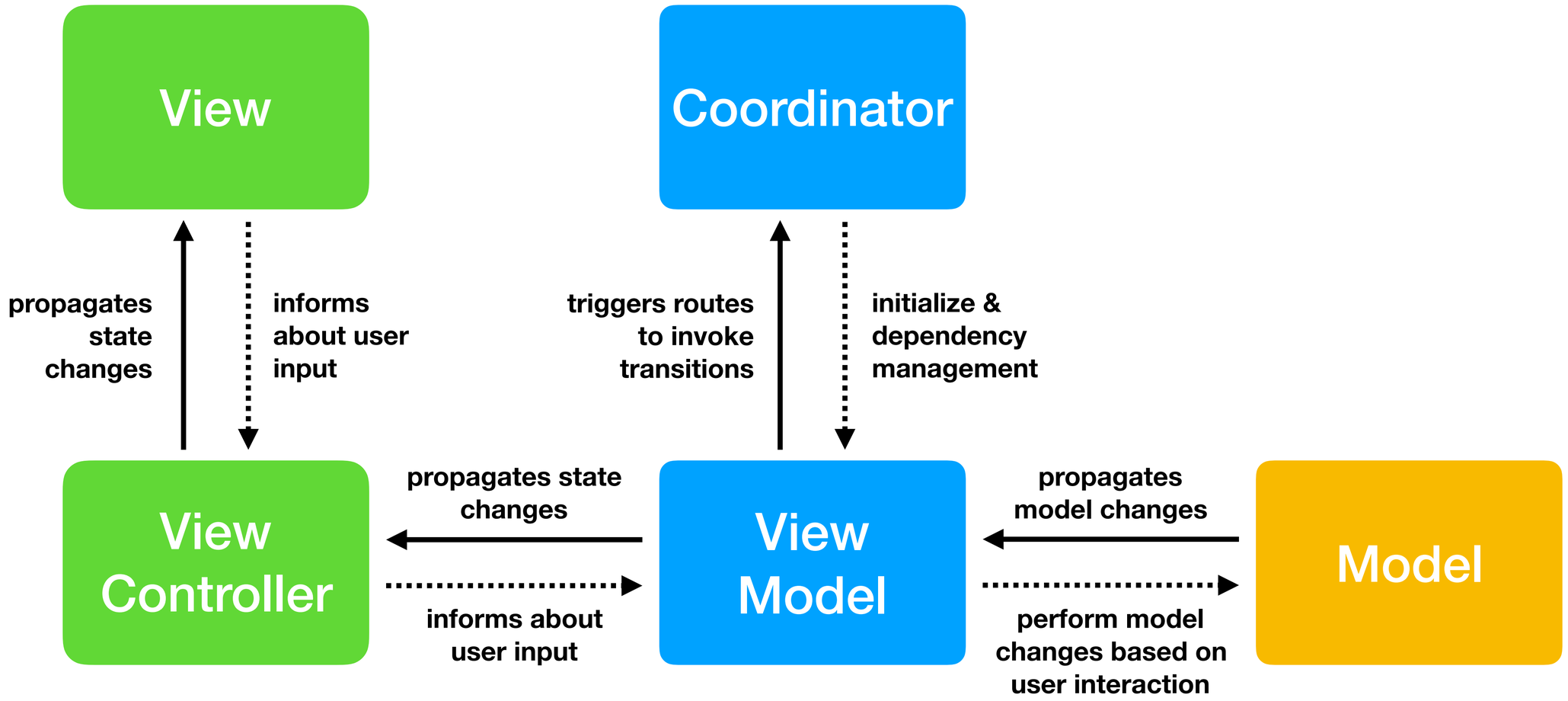
viewController-independent transitions

type-safety for transition destination

Coordinator pattern in MVC



Coordinator pattern in MVVM



Coordinator Pattern?

👍🏻 **Separation of concerns**

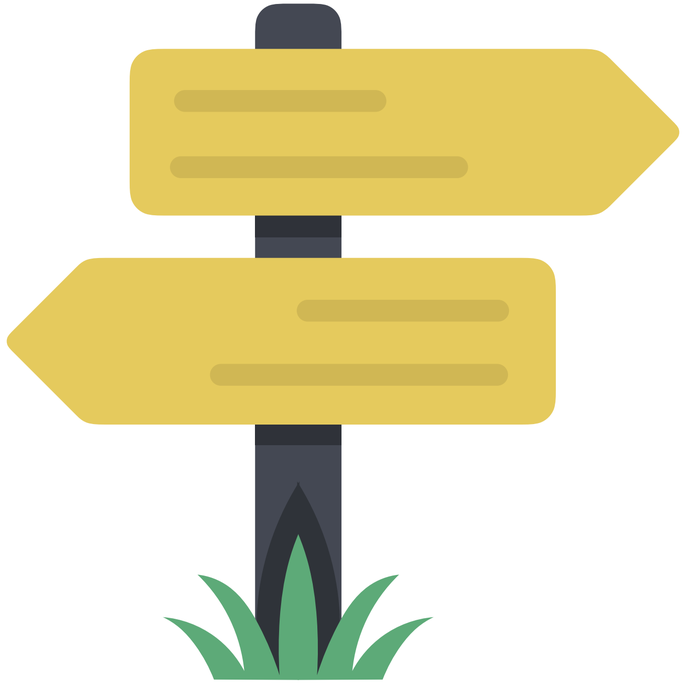
👍🏻 **Flow-independent, reusable scenes / view controllers**

👍🏻 **Easily changeable navigation logic**

👎🏻 **Programming overhead**

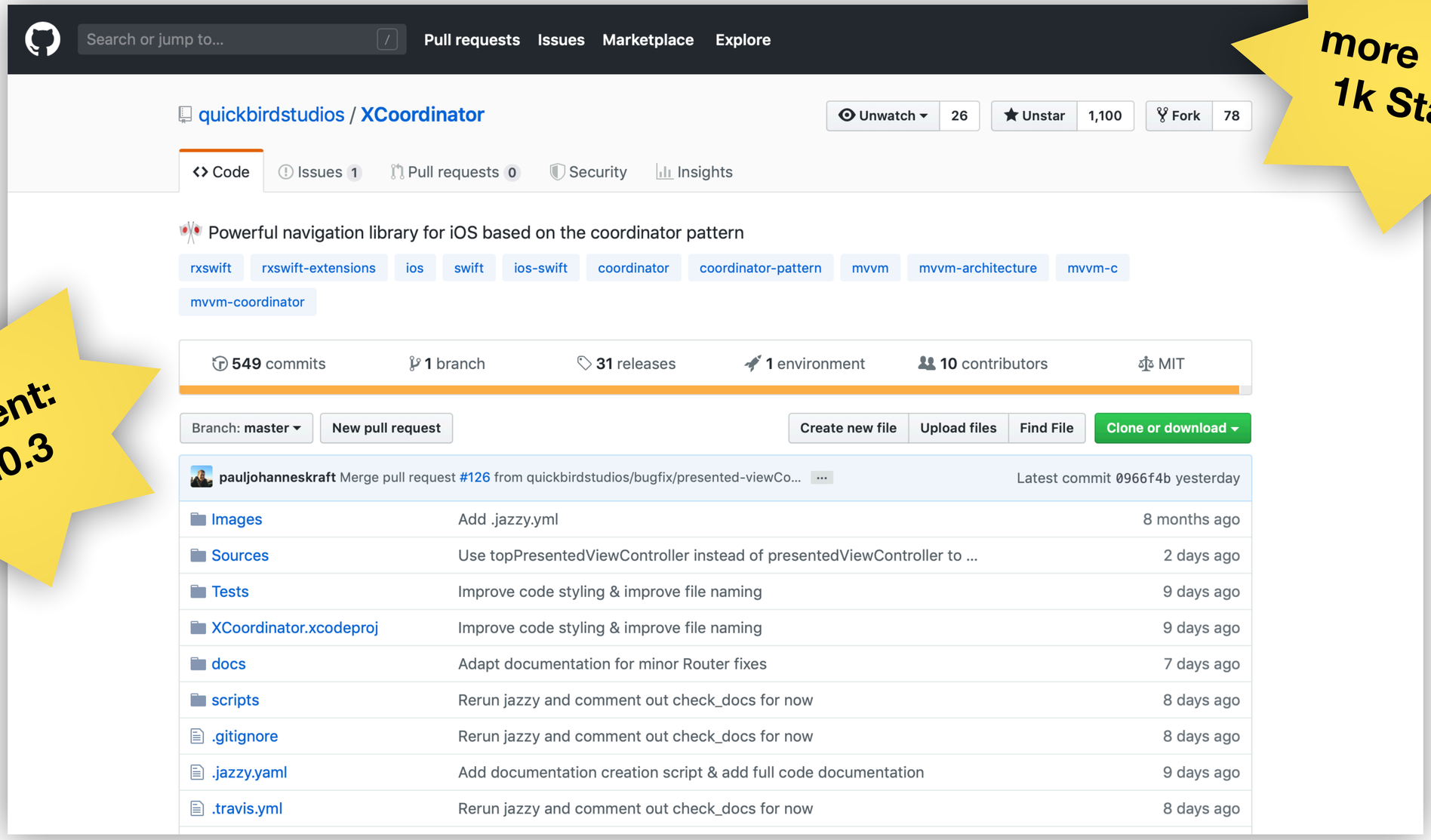
👎🏻 **Not the “standard way”**

And that’s how we got to build…



XCoordinator

XCoordinator is on GitHub!



more than 1k Stars

Current: v2.0.3

Apps we built with XCoordinator

soundperience LINOVA SOFTWARE GMBH

Soundfit Pro

ZEPPELIN CATERPILLAR

Operator Challenge

Lufthansa

Design Discovery

XSTUDY

Schultopf

Why use XCoordinator?

🌓 **Many different predefined transitions**

🏃‍♂️ **Simpler, consistent transition animation interface**

🔩 **Specialized coordinators for most use cases, generic BaseCoordinator**

👨‍✈️ **Fast switching of coordinators without changing viewController code**

🛤 **Deep Linking of routes of different types**

🚀 **RxSwift & Combine extensions**

 Route

Declaration:

```
enum LoginRoute: Route {
    case loginSuccessful
    case loginFailed
    case url(URL)
}
```

Usage (in ViewController or ViewModel):

```
coordinator.trigger(.loginSuccessful)
```

👨‍✈️ Coordinator

- Has a rootViewController
- Prepares transitions for triggered routes by initializing scenes
- performs transitions between scenes

```
class LoginCoordinator: ViewCoordinator<LoginRoute> {

    init() {
        let viewController = LoginViewController()
        super.init(rootViewController: viewController,
                   initialRoute: nil)
        viewController.router = unownedRouter
    }

    override func prepareTransition(
        for route: LoginRoute
        ) -> ViewTransition {
        /* … */
    }

}
```

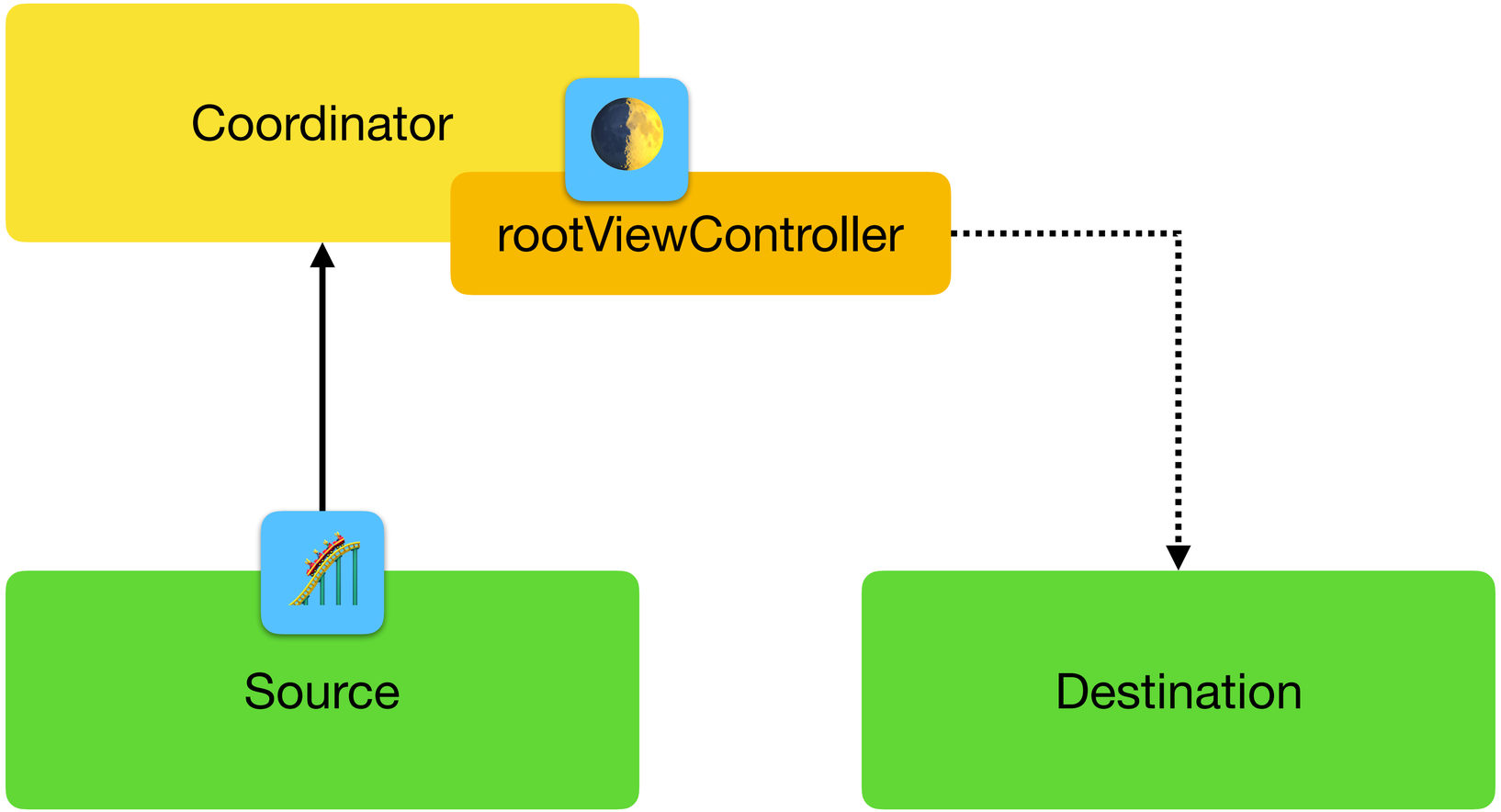
👨🏻‍✈️ Coordinator: Defining transitions

```
class LoginCoordinator: ViewCoordinator<LoginRoute> {
    /* … */

    override func prepareTransition(for route: LoginRoute) -> ViewTransition {
        switch route {
        case .loginSuccessful:
            let coordinator = HomeCoordinator()
           return .present(coordinator)
        case .loginFailed:
            let alert = createLoginFailedAlertController()
            return .present(alert)
        }
    }
}
```

👨‍✈️ Coordinator: Router Abstractions

- **Goals**
  - Remove context type information from Coordinator (i.e. which rootViewController type is used) —> Reusability, e.g. iPad / iPhone
  - Restrict Coordinator in ViewController/ViewModel to **trigger** method
  - Want to be able to hold a strong/weak/unowned reference to coordinators
- **Router Types**
  - **StrongRouter** (holds strong reference): Used in AppDelegate or for holding child coordinators
  - **UnownedRouter** (holds unowned reference): Used in ViewController/ViewModel or for holding parent coordinators
  - **WeakRouter** (holds weak reference): Used in ViewController/ViewModel or for holding parent coordinators - when coordinator might not exist anymore, when accessing it

How routes are triggered



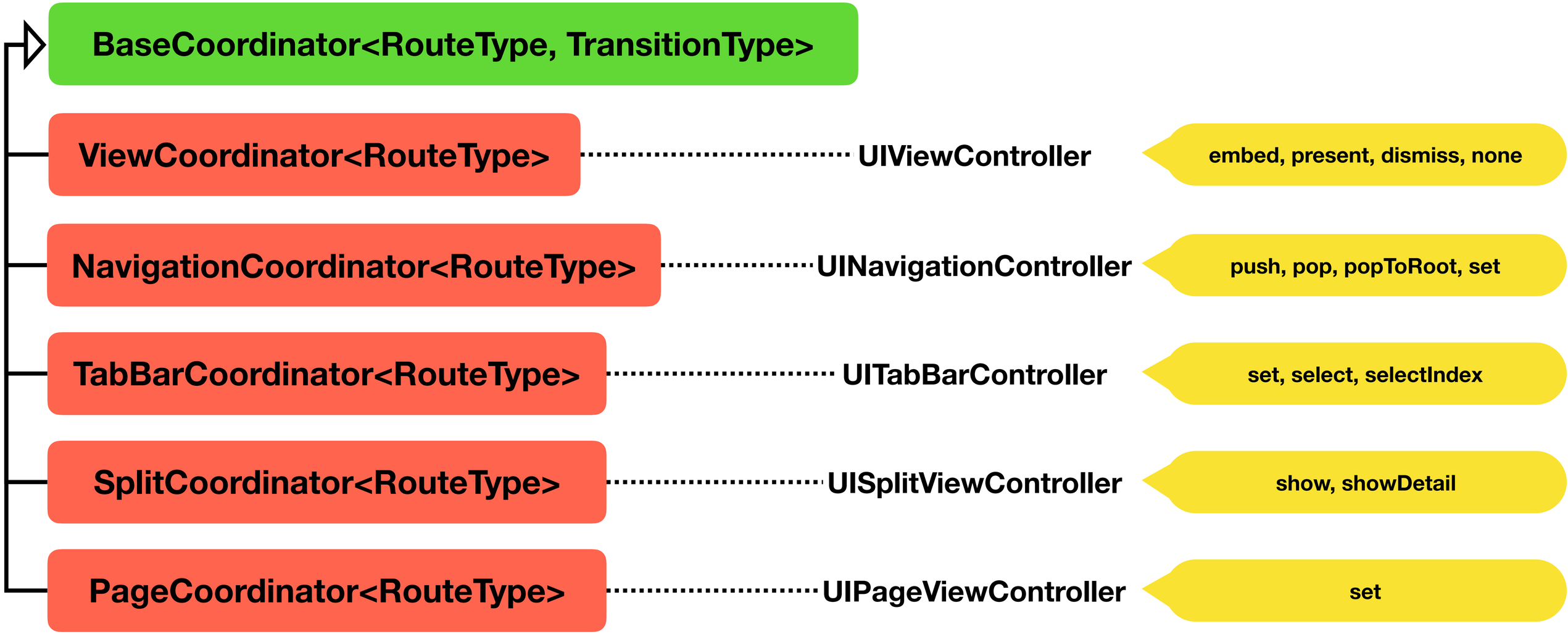
What is a Coordinator?

- **init(rootViewController:initialRoute:)**
- **trigger(_:with:completion:)**
- **prepareTransition(for:)**

What is a Coordinator?

- **init(rootViewController:initialRoute:)**
  - You can trigger a route right at initialisation time
  - Depending on the Coordinator type, there might be other options
  - Inject your custom rootViewController here - there is no possibility to change it afterwards.
- **trigger(_:with:completion:)**
  - Triggers a route, options can define to (not) animate the transition
- **prepareTransition(for:)**
  - Prepares transitions for a given route - might not be the same for different coordinator implementations of the same route

Coordinator & Transition types



Let’s integrate XCoordinator in our app!

Munich iOS

Introduce XCoordinator to your app

- **Create Route enum cases** for all possible segues or transition code segments
- **Add UnownedRouter** of the created Route-enum to the viewControllers handling transitions and replace the transition code / segues with triggering of routes
- **Implement a Coordinator** by overriding the prepareTransition(for:) method
- Make sure to **use the Coordinator** 😉

Exercise: Integrating XCoordinator

Task 1: Create LoginCoordinator & use as initial coordinator

Task 2: Create HomeCoordinator as TabBarCoordinator

Munich iOS

Solution 1: LoginCoordinator

Create **LoginRoute.swift** in **Routing**

- Create a LoginRoute enum with one case for each possible transition of that flow
- Introduce an `UnownedRouter<LoginRoute>` into `LoginViewController` and replace transition logic with triggering routes
- Create a LoginCoordinator as a ViewCoordinator
  - Override `generateRootViewController` to create a `LoginViewController` as the coordinator's rootViewController
  - Override `prepareTransition(for:)` to prepare transitions for the given routes
  - Create an empty initializer to make sure, the coordinator is always correctly initialized
- Use `coordinator.setRoot` to set the rootViewController of the app's window
- Try it out! Check, if everything still works.

Solution 2: HomeCoordinator

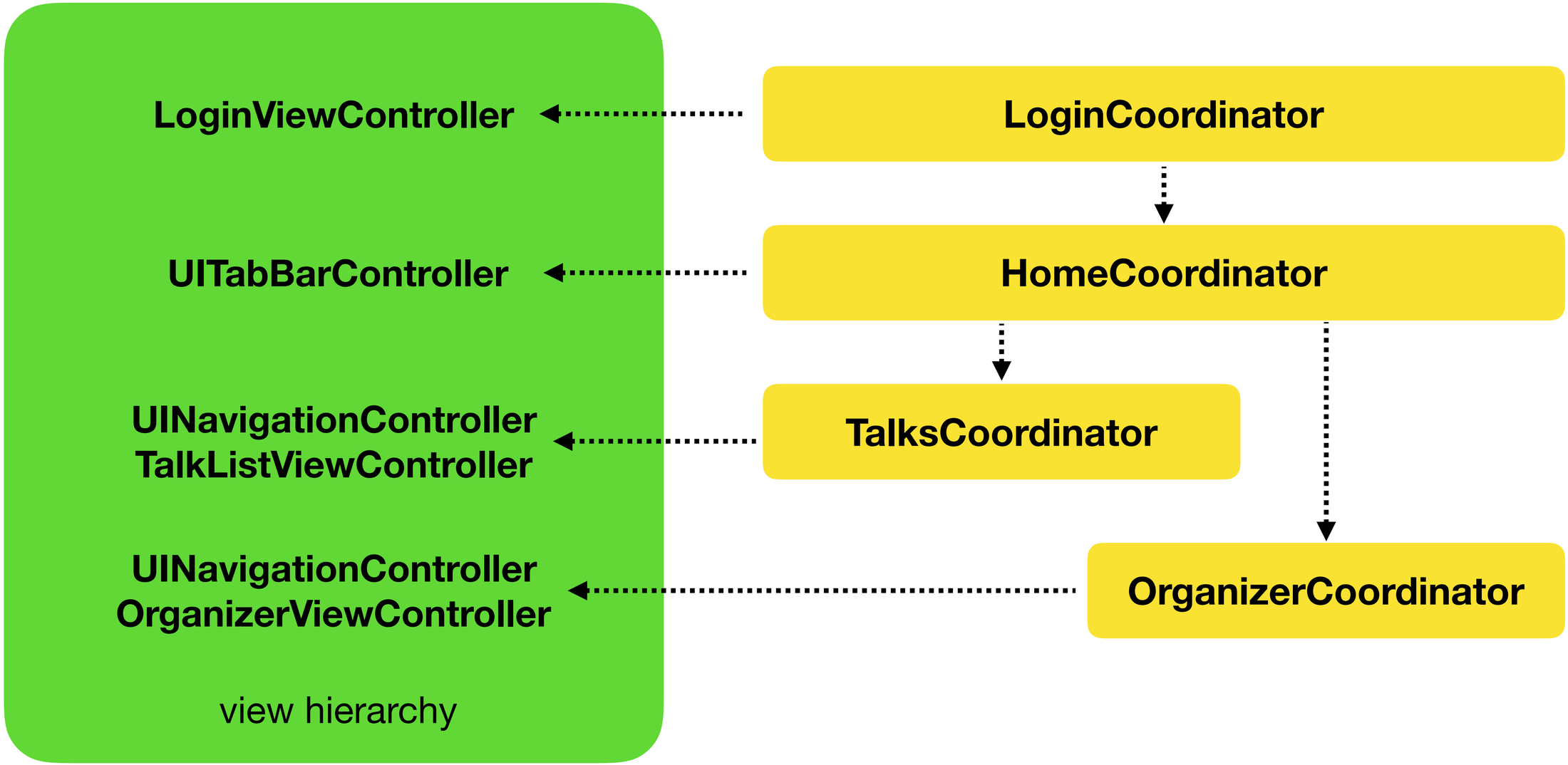
Create **HomeRoute.swift** in **Routing**

- Identify `HomeCoordinator` as a useful abstraction
- Create a `HomeRoute` with only an `initial` route, since there are no interactions possible
- Implement `HomeCoordinator` as a `TabBarCoordinator`
  - Create empty initializer to make sure it is always correctly initialized
  - Override `prepareTransition(for:)` to provide transitions for triggered routes
- Change `prepareTransition(for:)` in `LoginCoordinator` to present the coordinator instead of creating viewControllers

What did we achieve? 🎉

- Abstraction from transition logic & individual scenes/viewControllers
  - You can easily swap out, which transition is performed, when a route is triggered without the need of changing a viewController —> higher reusability
- Type-safe transitions (You cannot trigger a push transitions on a UITabBarController)
- Simpler transition animation interface
- Creation of scenes and connected model data at one place
  - You can also pass data using associated values in enum cases, if you want/need to pass data

Coordinator hierarchies



When do I need a new Route/Coordinator?

- **Rule of thumb:** new rootViewController to perform transitions on
- New Context
  - creation/editing of a model element
  - tabs in a UITabBarController that have distinct features
- Presentation of new Scene
- Restricting access of certain scenes to specific routes (see also **RedirectionRouter**)

Exercise: Finishing up integration

Task 1 (TalksRoute.swift): Introduce a TalksCoordinator

Task 2 (HomeRoute.swift): Make sure to highlight the Organizer-tab when routing to the Home route

Task 3 (OrganizerRoute.swift): Introduce an OrganizerCoordinator

Task 4 (LoginRoute.swift): Add login input validation

Solution 1.1 - Slide 1/2

- Create **TalksRoute** in **TalksRoute.swift**

```
enum TalksRoute: Route {
    case initial
}
```

Solution 1.1 - Slide 2/2

- Create **TalksCoordinator** in **TalksRoute.swift**

```
class TalksCoordinator: NavigationCoordinator<TalksRoute> {

    init() {
        super.init(initialRoute: .initial)
    }

    override func prepareTransition(for route: TalksRoute) -> NavigationTransition {
        switch route {
        case .initial:
            let viewController = TalkListViewController()
            viewController.talks = Model.createTalks()
            viewController.router = unownedRouter
            return .push(viewController)
        }
    }
}
```

Solution 1.2 - Slide 1/2

- Adapt **TalksRoute** in **TalksRoute.swift**

```
enum TalksRoute: Route {
    case initial
    case detail(Talk)
}
```

- Adapt **prepareTransition(for:)** in **TalksCoordinator**

```
case let .detail(talk):
    let viewController = TalkDetailViewController()
    viewController.talk = talk
    viewController.router = unownedRouter
    return .push(viewController, animation: .fade)
```

(optional) Solution 1.3

Solution 1.2 - Slide 2/2

- Add **router** property to **TalksViewController** in **TalksViewController.swift**

```
class TalkListViewController: UIViewController {
    /* .. */
    var router: UnownedRouter<TalksRoute>!
}
```

- Trigger detail route in **tableView(_:didSelectRowAt:)**

```
extension TalkListViewController: UITableViewDelegate {
    func tableView(_ tableView: UITableView,
                   didSelectRowAt indexPath: IndexPath) {
        let model = talks[indexPath.row]
        router.trigger(.detail(model))
    }
}
```

Solution 2

- Use **TalksCoordinator** by adapting **HomeCoordinator** in **HomeRoute.swift**

```
override func prepareTransition(for route: HomeRoute) -> TabBarTransition {
    switch route {
    case .initial:
        let organizerViewController = OrganizerViewController()
        organizerViewController.organizer = Model.createOrganizer()
        let orgaRoot = UINavigationController(rootViewController: organizerViewController)
        let talksCoordinator = TalksCoordinator()
        return .multiple(
            .set([talksCoordinator, orgaRoot]),
            .select(orgaRoot)
        )
    }
}
```

Solution 3.1

- Create **OrganizerRoute** in **OrganizerRoute.swift**

```
enum OrganizerRoute: Route {
    case initial
}
```

Solution 3.1

- Create **OrganizerRoute & OrganizerCoordinator** in **OrganizerRoute.swift**

```
enum OrganizerRoute: Route {
    case initial
}


class OrganizerCoordinator: NavigationCoordinator<OrganizerRoute> {

    init() {
        super.init(initialRoute: .initial)
    }

    override func prepareTransition(for route: OrganizerRoute) -> NavigationTransition {
        switch route {
        case .initial:
            let viewController = OrganizerViewController()
            viewController.organizer = Model.createOrganizer()
            return .push(viewController)
        }
    }
}
```

(Optional) Solution 3.2

- Add **website(URL)** case to **OrganizerRoute**

```
enum OrganizerRoute: Route {
    case initial
    case website(URL)
}
```

- Adapt **prepareTransition(for:)** in **OrganizerCoordinator**

```
case let .website(url):
    UIApplication.shared.open(url)
    return .none()
```

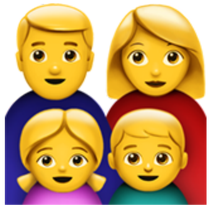
(Optional) Solution 3.2

- Trigger **.website(URL)** in **OrganizerViewController**

```
var router: UnownedRouter<OrganizerRoute>!


@objc private func websiteButtonTapped() {
    guard let url = organizer?.website,
        UIApplication.shared.canOpenURL(url) else {
        return assertionFailure()
    }

    router.trigger(.website(url))
}
```

Don’t forget to set the **router** of the **OrganizerViewController** in the **OrganizerCoordinator**!

Now let’s explore some more advanced features…



Connections between coordinators

Deep Linking

Reactive extensions

Exercise 2: Adding more functionality

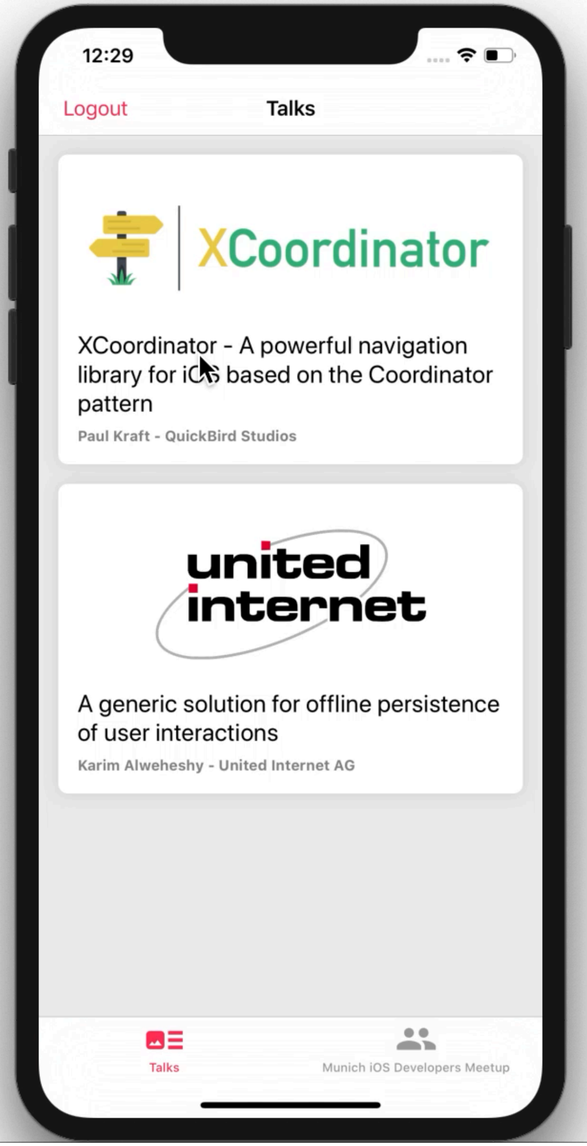
Task 1: Add “Logout” functionality using parent/child coordinators

Task 2: Open URLs using Deep Linking

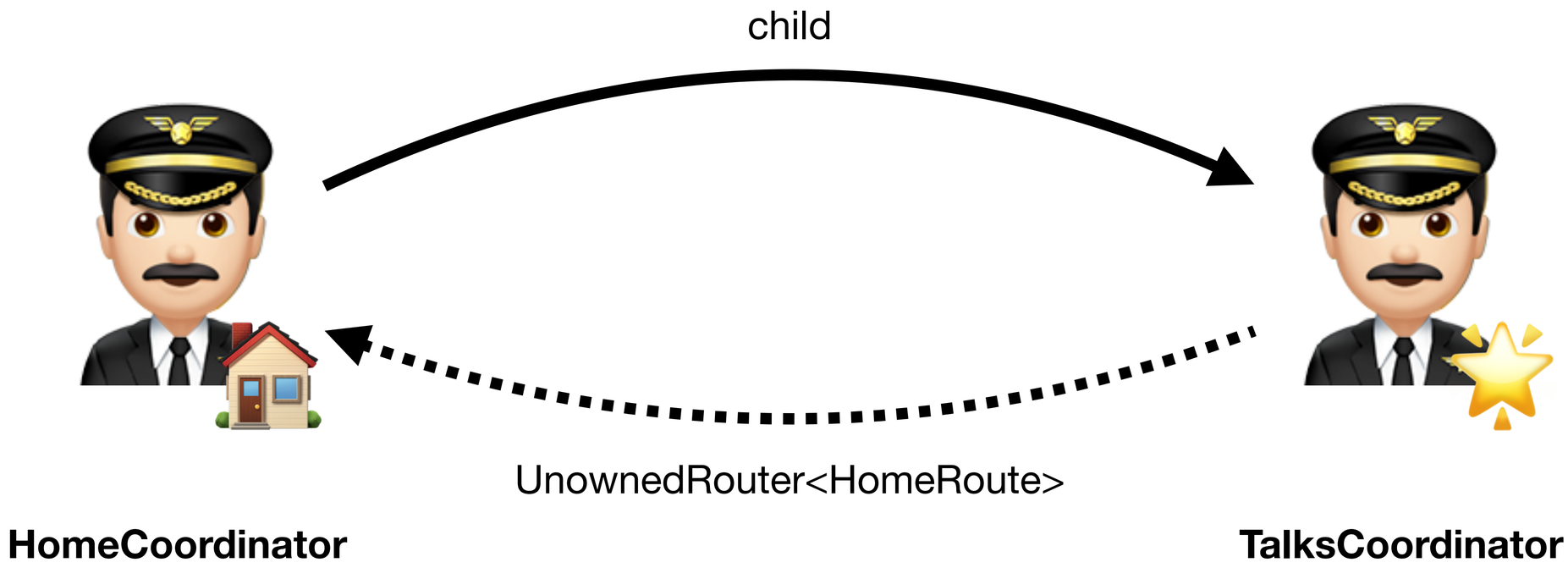
Task 3: Login with reactive streams (RxSwift / Combine)

Task 2.1: Logout

- Dismiss the **UITabBarController** on tap of “Logout” button
- Handle the transition in **HomeCoordinator**, even though **TalkListViewController** only has reference to **TalksCoordinator**
- Keep unowned reference of **HomeCoordinator** in **TalksCoordinator**
- Adapt **prepareTransition**-methods on **HomeCoordinator** and **TalksCoordinator**
- trigger **TalksRoute.logout** in **TalkListViewController**

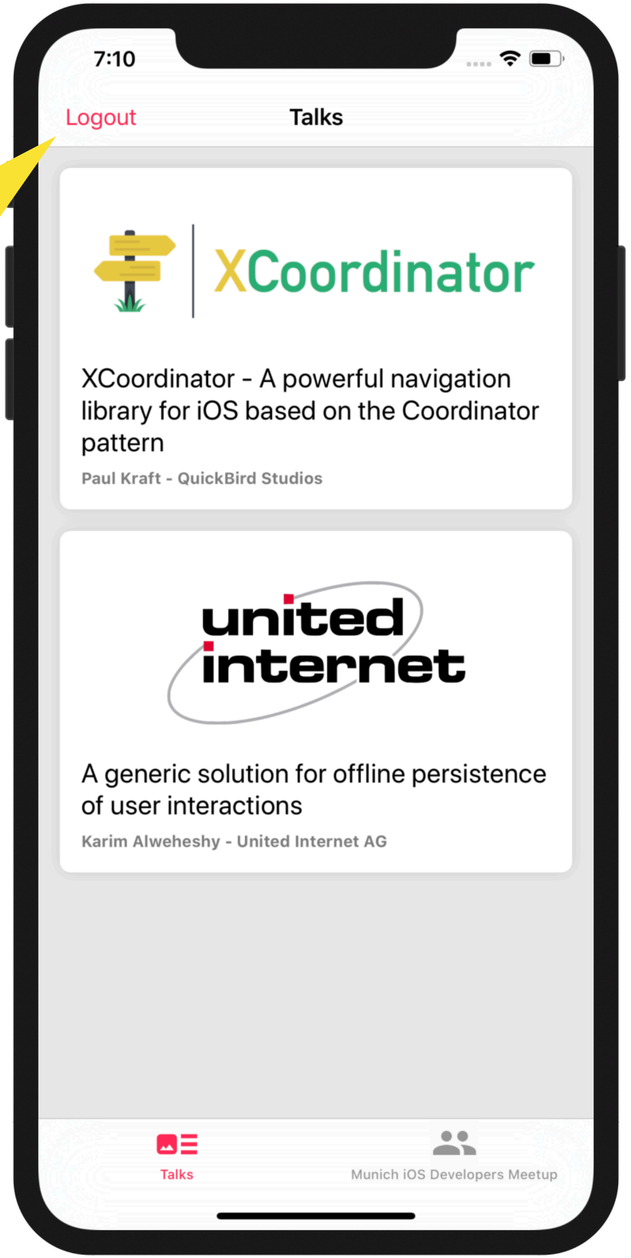


Task 2.1: 👨‍👧 Parent/Child coordinators



Task 2.1: Logout

return to “Login” screen

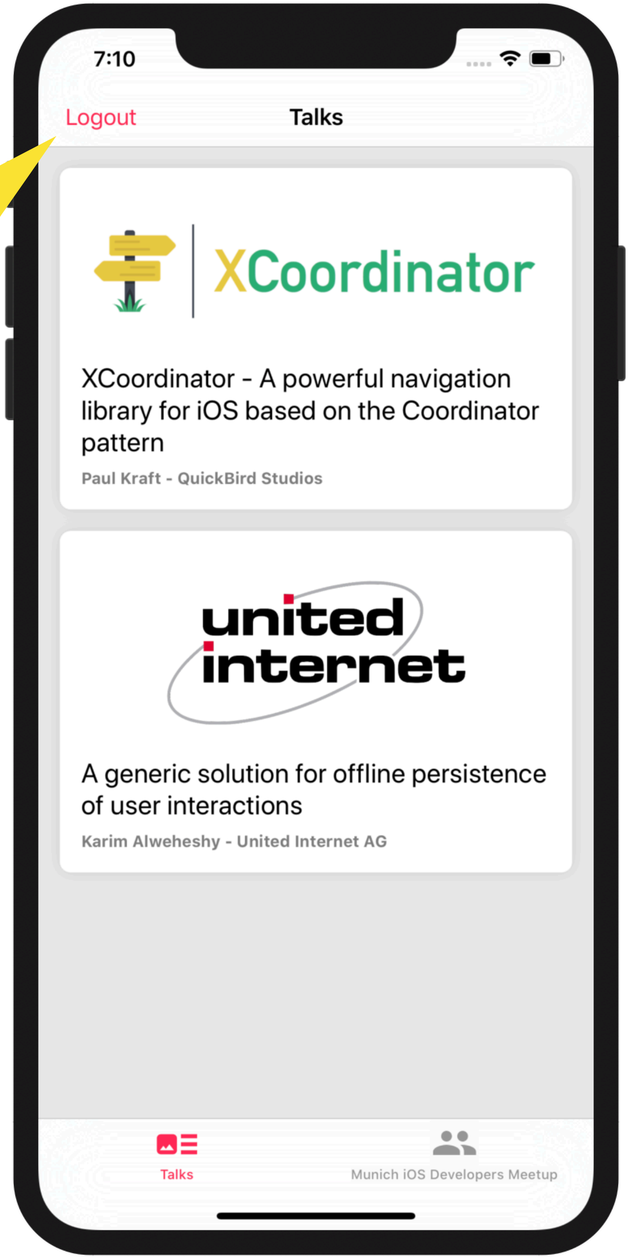


- Add UITabBarItem to TalksViewController with according action

```
private lazy var logoutButton =
    UIBarButtonItem(title: "Logout", style: .plain,
                    target: self, action: #selector(didTapLogout))

@objc private func didTapLogout() {
    router.trigger(.logout)
}
```

Task 2.1: Logout

return to “Login” screen

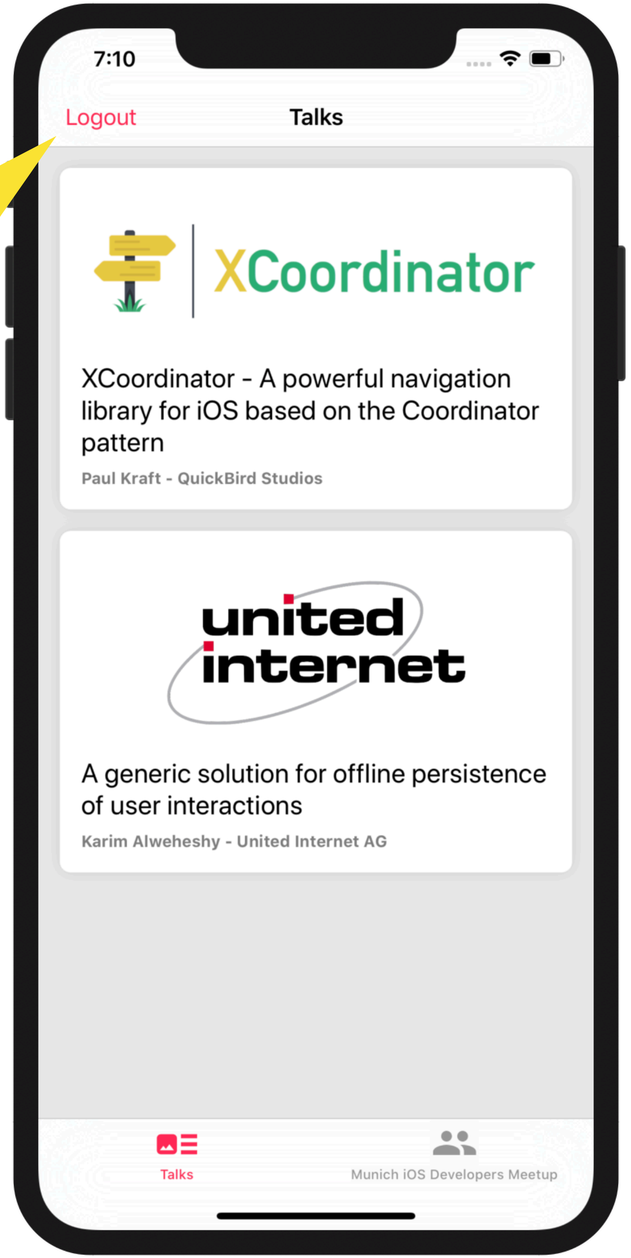


- Implement route transition in **TalksRoute.swift**

```
private let parent: UnownedRouter<HomeRoute>

init(parent: UnownedRouter<HomeRoute>) {
    self.parent = parent
    super.init(initialRoute: .initial)
}

override func prepareTransition(for route: TalksRoute
    ) -> NavigationTransition {

    switch route {
    /* … */
    case .logout:
        return .trigger(.logout, on: parent)
    }
}
```

Task 2.1: Logout

return to “Login” screen



- Implement route transition in **HomeRoute.swift**

```
private lazy var talksRouter =
    TalksCoordinator(parent: unownedRouter).strongRouter


override func prepareTransition(for route: HomeRoute
    ) -> TabBarTransition {
    switch route {
    /* … */
    case .logout:
        return .dismiss()
    }
}
```

Exercise 2: Adding more functionality

Task 1: Add “Logout” functionality using parent/child coordinators

Task 2: Open URLs using Deep Linking

Task 3: Login with reactive streams (RxSwift / Combine)

Task 2.2: 🛤 Deep Linking

- Let’s assume your app has not been started the usual way…

URL

Notification

State restoration

- How can you navigate through large parts of your app at once?

Task 2.2: Deep Linking

- We will inject the url in the SceneDelegate (it would normally be triggered by another app)

ios-meetup-munich:talk?name=XCoordinator

- The app should open with the talk visible at startup
- We will define a **LoginRoute.url(URL)** and use it as the initialRoute in the LoginCoordinator
- The LoginCoordinator already has a **talk(from:)** method to extract a talk from the url, if available.

Task 2.2: Opening URLs

- Use the predefined talk(from:) method in **LoginRoute.swift**

```
override func prepareTransition(for route: LoginRoute) -> ViewTransition {
    switch route {
    /* … */
    case .url(let url):
        guard let talk = self.talk(from: url) else {
            return deepLink(.loginSuccessful, HomeRoute.talks)
        }
        return deepLink(.loginSuccessful, HomeRoute.talks, TalksRoute.detail(talk))
    }
}
```

- When using deep linking, make sure to include transitions to all coordinators along the coordinator hierarchy, since otherwise, the deep link will fail at runtime.

Exercise 2: Adding more functionality

Task 1: Add “Logout” functionality using parent/child coordinators

Task 2: Open URLs using Deep Linking

Task 3: Login with reactive streams (RxSwift / Combine)

Task 2.3: 🚀 Reactive extensions

- **trigger** completion block informs about completed transitions
- **trigger** reactive extension returns Observable/Publisher
- Useful for triggering routes after certain reactive event happened (e.g. login)
- Useful for executing code after transition has completed

Regular:
```
router.trigger(.loginSuccessful, completion: { /* done */ })
```

RxSwift:
```
router.rx.trigger(.loginSuccessful) // Observable<Void>
```

Combine:
```
router.publishers.trigger(.loginSuccessful) // Future<Void, Never>
```

Task 2.3: Reactive extensions (RxSwift)

- API:

```
router.rx.trigger(<some route>)
```

- Example:

```
let loginObservable = login() // Observable<LoginData>
    .flatMap { [unowned self] loginData in
        self.router.rx.trigger(.loginSuccess(loginData))
    }
    .catchError { [unowned self] error in
        self.router.rx.trigger(.loginFailure(error))
    }
```

Task 2.3: Reactive extensions (Combine)

- API:

```
router.publishers.trigger(<some route>)
```

- Example:

```
let loginPublisher = login() // AnyPublisher<LoginData, Error>
    .flatMap { [unowned self] in
        self.router.publishers
            .trigger(.loginSuccessful)
            .mapError { _ -> Error in }
    }
    .catch { [unowned self] _ in
        self.router.publishers.trigger(.loginFailed)
    }
```

XCoordinator

QUICKBIRD STUDIOS

- Powerful navigation library for iOS using the **Coordinator pattern**
- Developed by **QuickBird Studios**, located in Munich
- Provides **base classes for different coordinator types**, such as **NavigationCoordinator, TabBarCoordinator,** and many more
- **Encapsulates navigation code** for UIKit and provides **consistent API for different transition types**
- **Predefined transition types** with completion handler to ensure that transitions are executed sequentially
- **RxSwift** & **Combine** extensions!
- Full support for **custom transitions & animations**

References

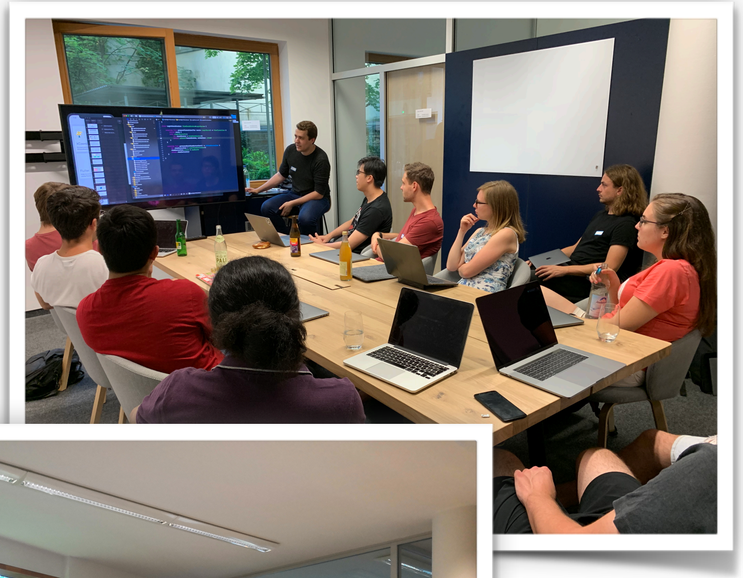
- **XCoordinator**: Powerful navigation library for iOS based on the coordinator pattern (https://github.com/quickbirdstudios/XCoordinator)

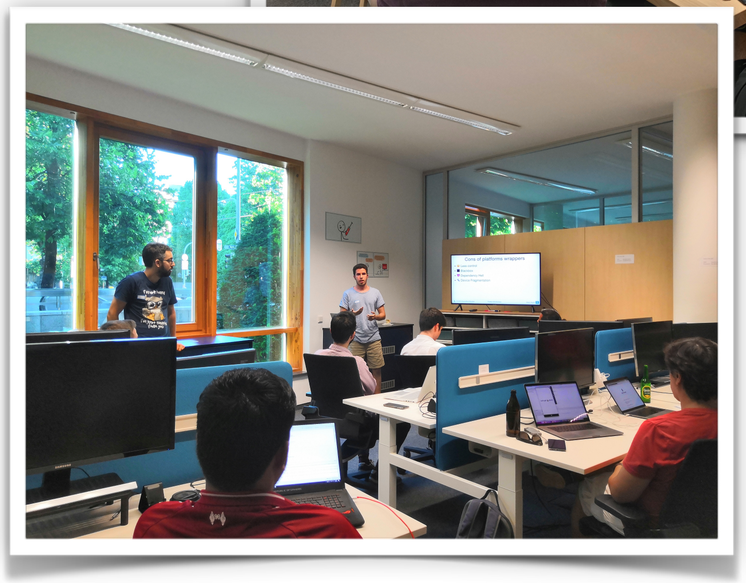
- **Introducing an iOS navigation library based on the coordinator pattern** by Stefan Kofler (https://quickbirdstudios.com/blog/ios-navigation-library-based-on-the-coordinator-pattern)

- **The Coordinator** by Soroush Khanlou (http://khanlou.com/2015/01/the-coordinator)

Mobile Hack Night @ QuickBird Studios

- 2 exciting workshops (at the same time):
  - SwiftUI: Get hands-on experience with Apple's newest UI framework for building iOS Apps
  - Mastering Gradle - How to tame the beast with buildSrc
- Presentation, Live Coding, Hacking, Socializing & Food!
- Tue, 5 November 2019 at 6:45 PM - 9:00 PM
- Location:
  - QuickBird Studios
  - Nymphenburger Str. 13-15
  - 80335 München
- Register here: bit.ly/mobileHack





App Advice

- You have questions about your app's architecture, mobile technologies or code style?
- 30 Minutes of free personal development advice from our experienced software engineers
- You can ask us about code snippets, scribble things on a blackboard and ask us many questions!
- Submit your request here: bit.ly/appAdvice30

QuickBird Studios

- **Mobile HackNight**
  - November 5, 6:45 PM - 9 PM, Nymphenburger Str. 13-15
  - Presentation + Live Coding + Hacking + Socializing + Food!
  - Topic: “SwiftUI” or “Mastering Gradle”
  - Registration: bit.ly/mobileHack
- **App Advice**
  - Free 30 Minutes of software development advice from our engineers
  - Ask us about your app’s architecture, mobile technologies or code style
  - Request here: bit.ly/appAdvice30