Thursday, 22 Nov. 2018 - 19h

Coding Testable Apps On Steroids

I wonder where the right building is

Architecture

You find great stuff on the Internet

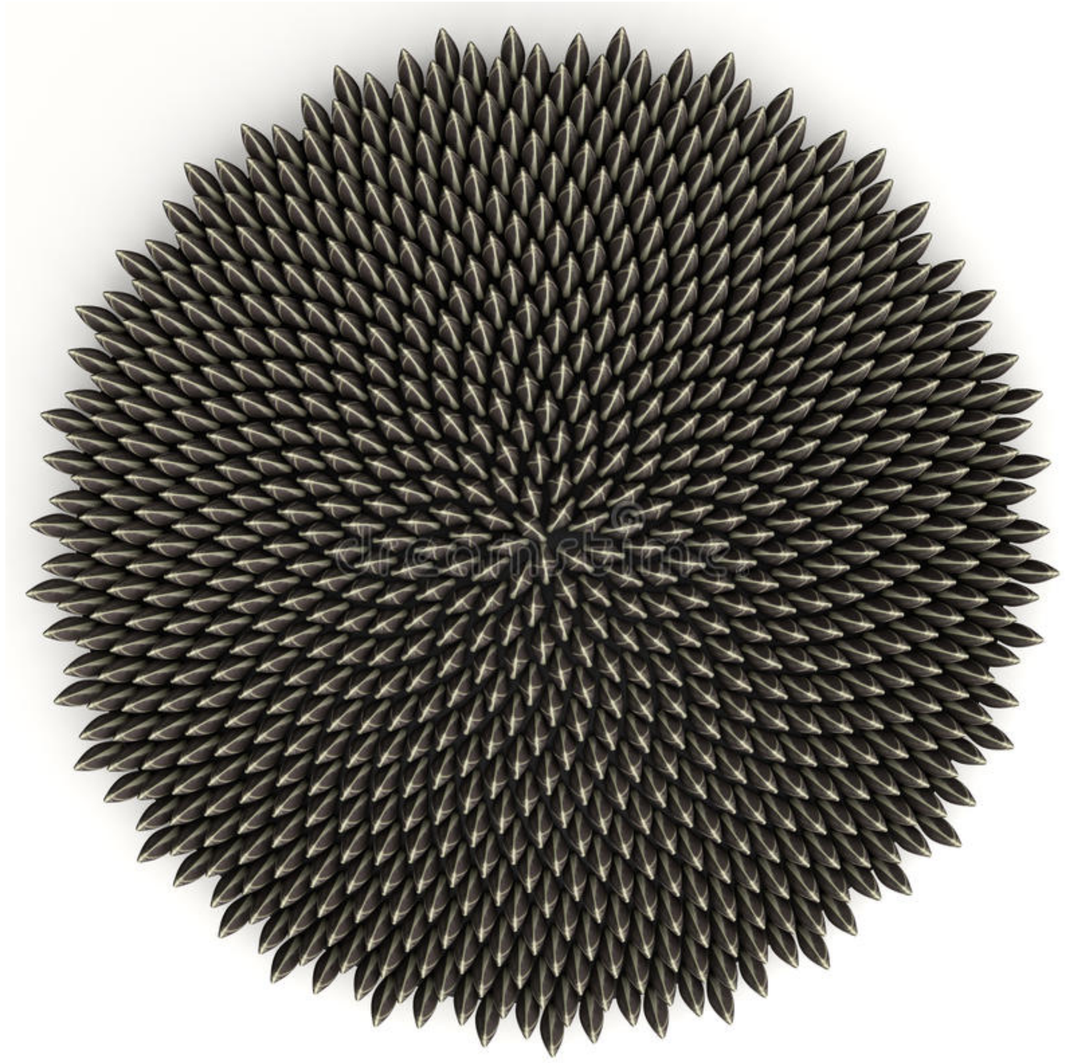
This too

The right house is really weird

Architecture

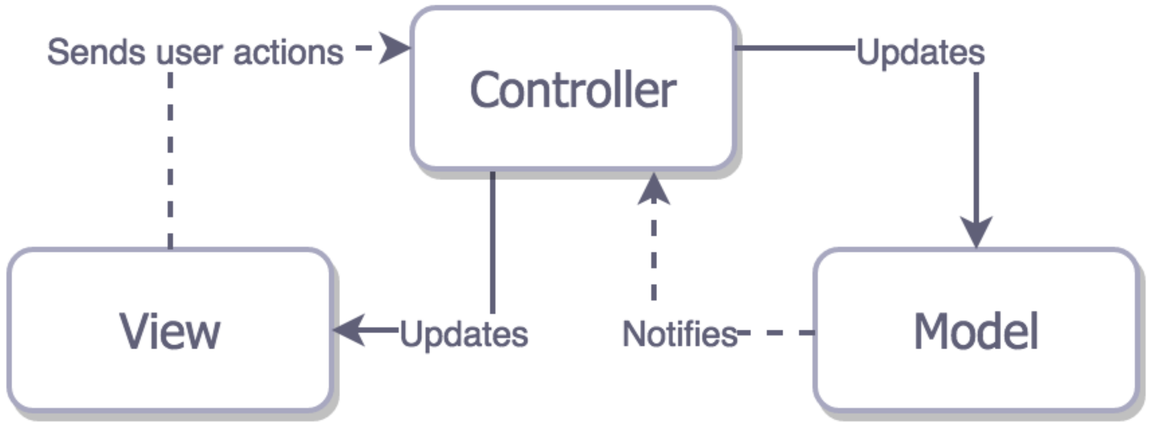
You find great stuff on the Internet

This

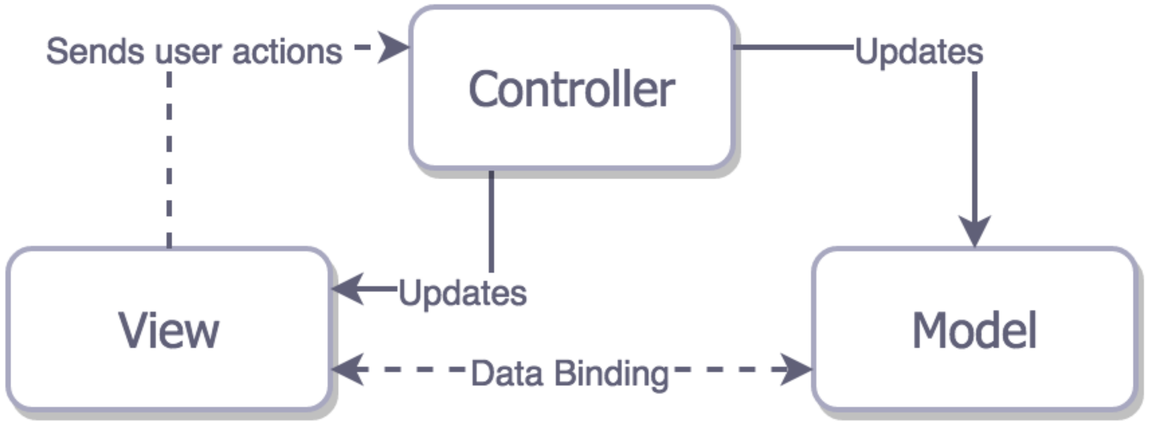


Architect ureP atter ns

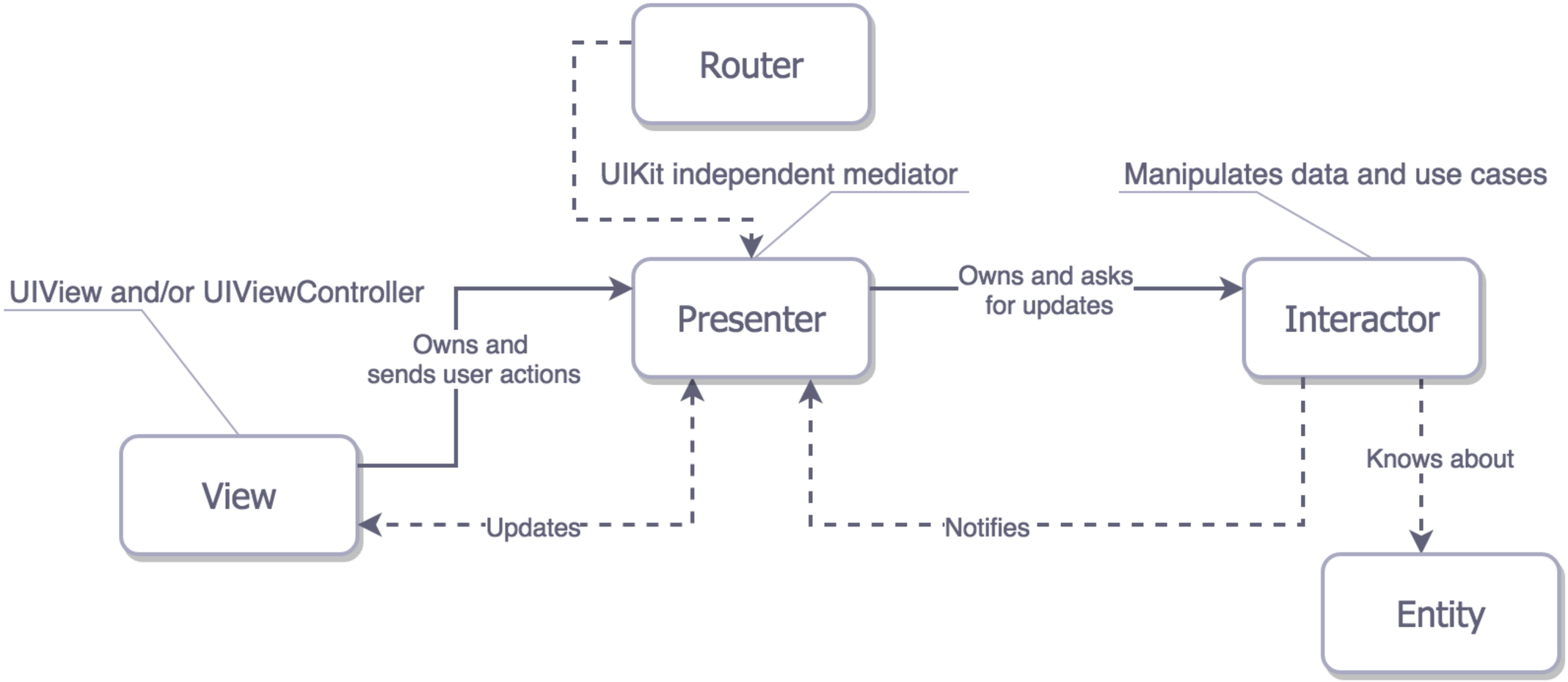
MVC


Sends user actions
Controller
Updates
View
Updates
Notifies
Model

MVP


Sends user actions
Controller
Updates
Updates
View
Model
Data Binding

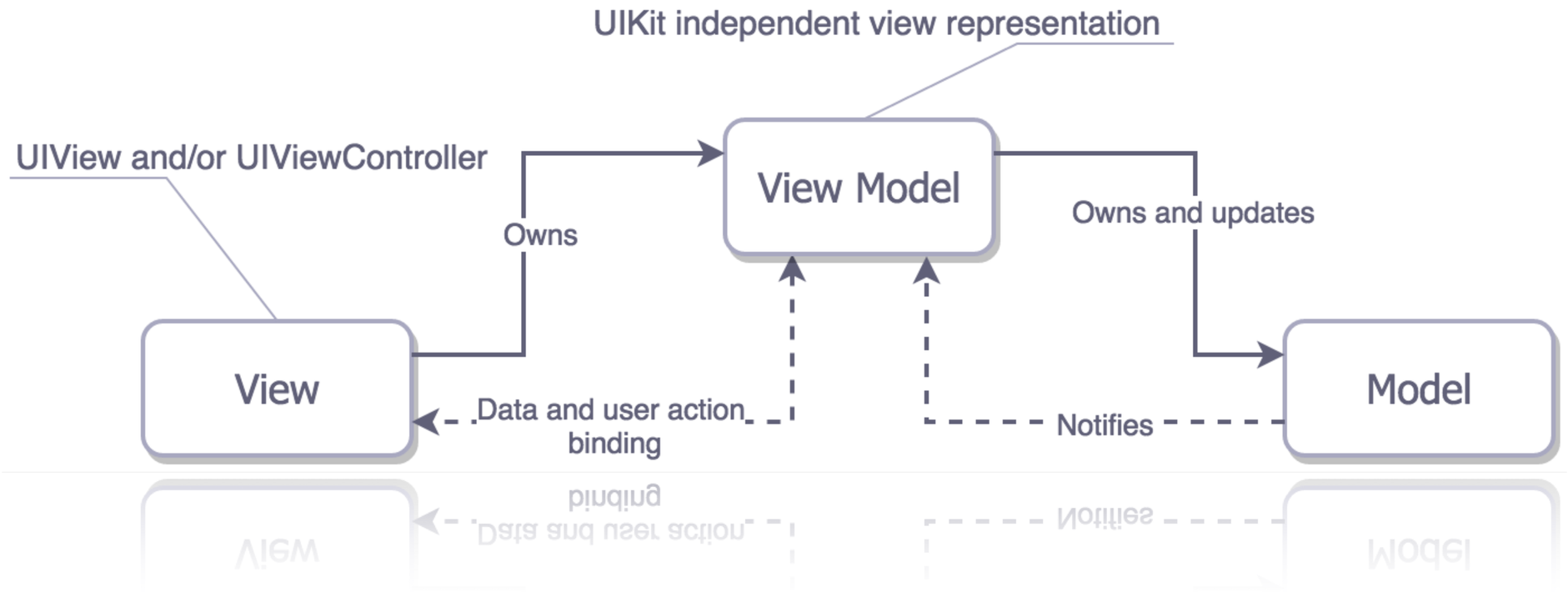
VIPER


Router
UIKit independent mediator
Manipulates data and use cases
UIView and/or UIViewController
Owns and
sends user actions
Presenter
Owns and asks
for updates
Interactor
View
Updates
Notifies
Knows about
Entity

We have a winner!

M V V M

MVVM



Hamburg had a better Spring...

Great city - Shitty Winter

Driver Tribe Barcelona

Stefan B.

Mounir D.

David C.

Carlos N.

Fabio C.

Adrian Z.

Ferran P.

Lluis G.

Working in a team

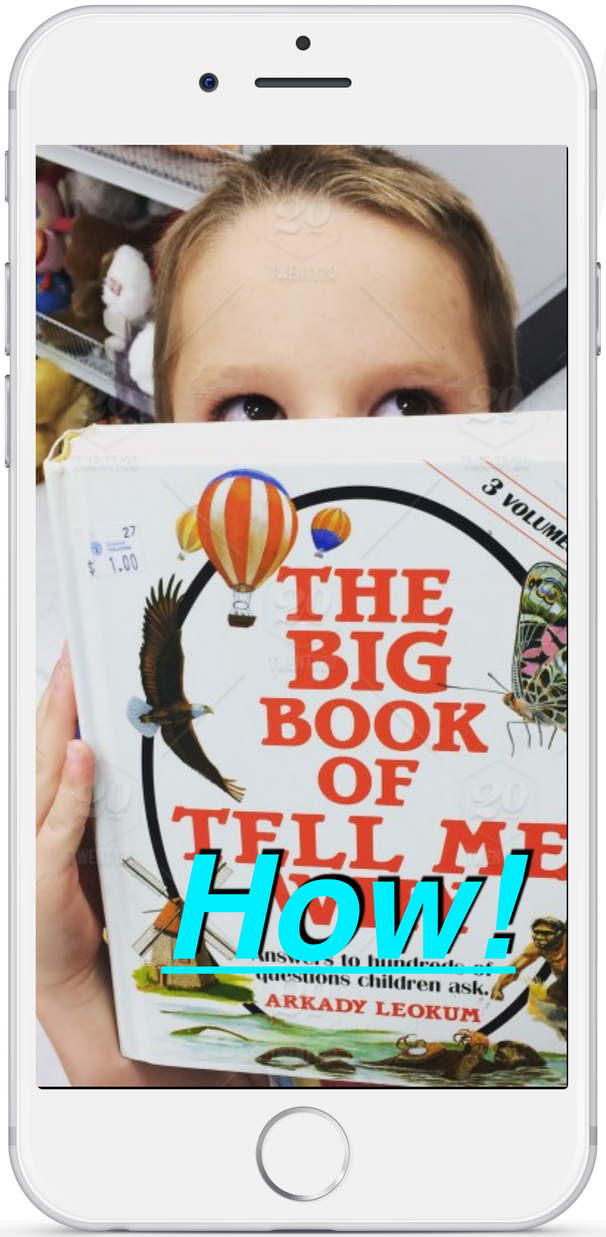
Everybody has different backgrounds

The work flow has to be aligned

Our architecture has to be well defined

Everyone should be able to bring in ideas

MVVM IS GREAT, BUT...



How exactly will our MVVM architecture look like?


travelapp
Username
Password
Login
Dont have an account? Register
iPhone XR - 12.1

Demo App

Contains the business logic ✅

Handles the current state of the app

Is easily testable

Consumes the user input

ViewModel

View

Is the view to the world ✅

Represents the current state of the app

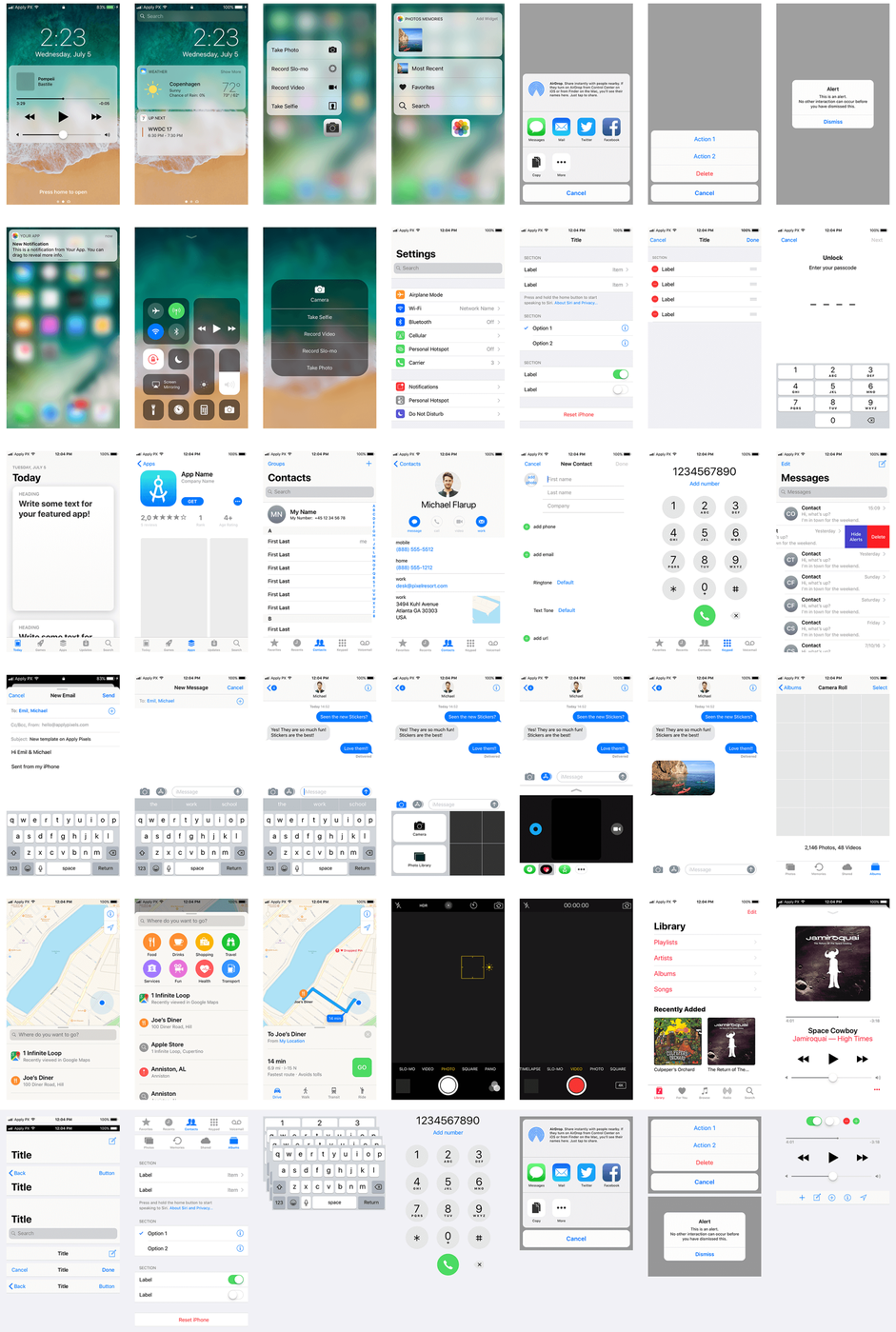
Should not contain any logic

Passes user input to view model

Questions

1

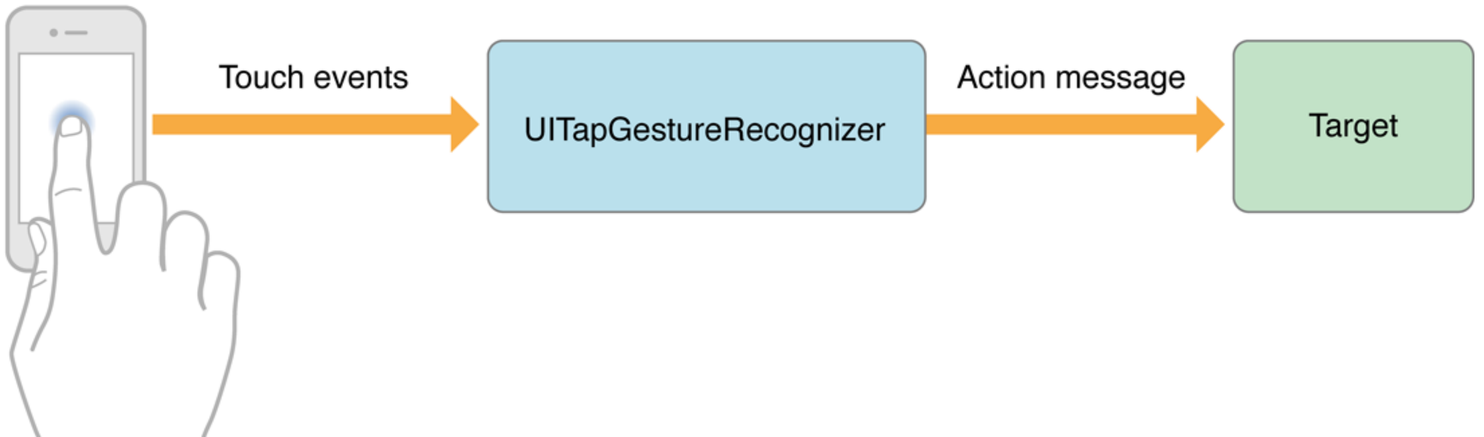
How to pass the user input to the view model?



Gestures

TOUCH (TAP)	SWIPE LEFT	DRAG	PINCH OPEN	SCROLL DOWN
LONG PRESS	SWIPE RIGHT	DOUBLE TOUCH	PINCH CLOSE	SCROLL UP

Tapping gesture



iOS Input Events are Asynchronous

Target-Action

Delegate-Pattern

Notification Pattern

Block-Based API (Closure)

Contains the business logic

Knows the current state of the app

Is easily testable

Consumes the user input

ViewModel

View

Passes user input to view model

: Action

sed API

Events

bserver

Be a D

Reactive Programming

Rx Swift

I just love it

Great Library

RxCocoa

Unifies all asynchronous events into observable events

Allows to bind to and from UI Elements

Allows operations to be performed on observable events

- UIBarButtonItem
- UIButton
- UICollectionView
- UIControl
- UIDatePicker
- UIImageView
- UILabel
- UIScrollView
- UISearchBar
- UISlider
- UITableView
- UITextField

Inputs that do the same thing can be merged

The View (Interface)

Code finally...

```
import RxCocoa
import RxSwift

final internal class LoginView : UIView {
   internal struct Model : Equatable {

        let isLoginButtonEnabled: Bool
        let isPasswordHidden: Bool
        let isSpinning: Bool
        let state: LoginViewModel.State
    }

    internal struct Outputs {

        let usernameField: Signal<String>
        let passwordField: Signal<String>
        let loginButton: Signal<Void>
        let registerButton: Signal<Void>
        let showPasswordButton: Signal<Void>
    }

    lazy var outputs: Outputs { get set }
}

extension Reactive where Base : LoginDemoRxRedux.LoginView {
    var inputs: Binder<LoginView.Model> { get }
}
```

The View (Implementation)

Code finally...

```
import RxCocoa
import RxSwift

final class LoginView: UIView {
    struct Outputs {
        let usernameField: Signal<String>
        let passwordField: Signal<String>
        let loginButton: Signal<Void>
        let registerButton: Signal<Void>
        let showPasswordButton: Signal<Void>
    }

    lazy var outputs: Outputs = {
        return LoginView.Outputs(usernameField: usernameTextField.rx.text.orEmpty.asSignal().distinctUntilChanged(),
                                 passwordField: passwordTextField.rx.text.orEmpty.asSignal().distinctUntilChanged(),
                                 loginButton: loginButton.rx.tap.asSignal(),
                                 registerButton: registerButton.rx.tap.asSignal(),
                                 showPasswordButton: showPasswordButton.rx.tap.asSignal())
    }()
}

extension Reactive where Base: LoginView {
    var inputs: Binder<LoginView.Model> {
        return Binder(self.base) { view, loginState in
            view.configure(from: loginState)
        }
    }
}
```

View Model Interface

Code finally...

```
import RxFeedback
import RxSwift
import RxCocoa

internal struct LoginStore {

    init(inputs: LoginView.Outputs)

    lazy var output: Driver<LoginView.Model> { mutating get set }

    enum State : Hashable {

        case loggedOut, loggedIn(User), loginFailed(APIError), performingLogin
    }
}
```

Contains the business logic ✅

Knows the current state of the app

Is easily testable

Consumes the user input ✅

ViewModel

View

Is the view to the world ✅

Represents the current state of the app

Should not contain any logic

Passes user input to view model ✅

Questions

2

How to handle the state of the app?

Best game to demo states

Street Fighter 2

What is state?

It should be at one place

State is everywhere

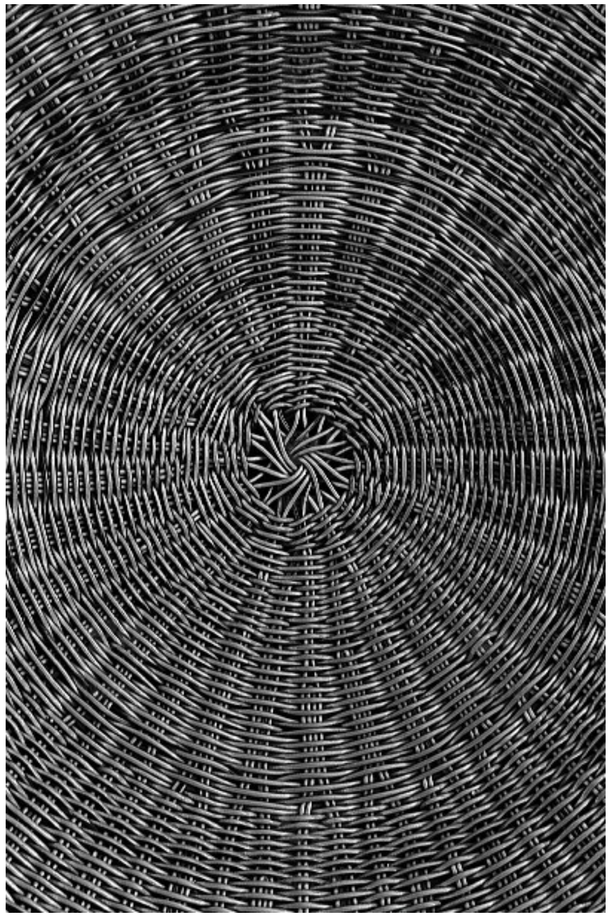
State Problems

Usually state is everywhere

There are more states than needed

Hard to visualize the different states

You must be careful not to be in two different states at the same time



The TAMING of the SHREW

By WILLIAM SHAKESPEARE

Edited by BARBARA A. MOWAT
and PAUL WERSTINE

Swift is great

```
enum State: Hashable {
      case loggedOut
      case loggedIn
      case loginFailed
      case performingLogin
 }
```

Model State as Enum

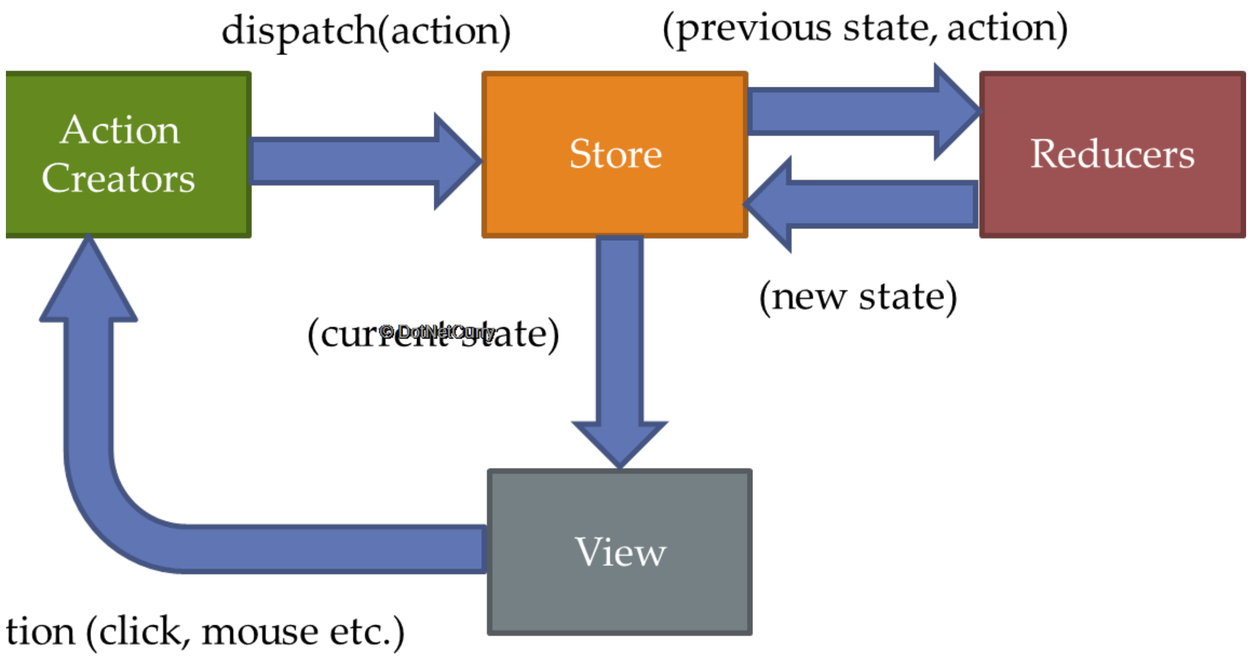
¿Qué es **Redux**?

New State

Input from UI Interface

Ugly image found on the web

Redux flow



Reducers are pure functions

What is a reducer?

It should be at one place

State is tamed

State Handling

Unidirectional Dataflow

Store takes input and produces an output. Control flow is unidirectional

Immutable Value Types

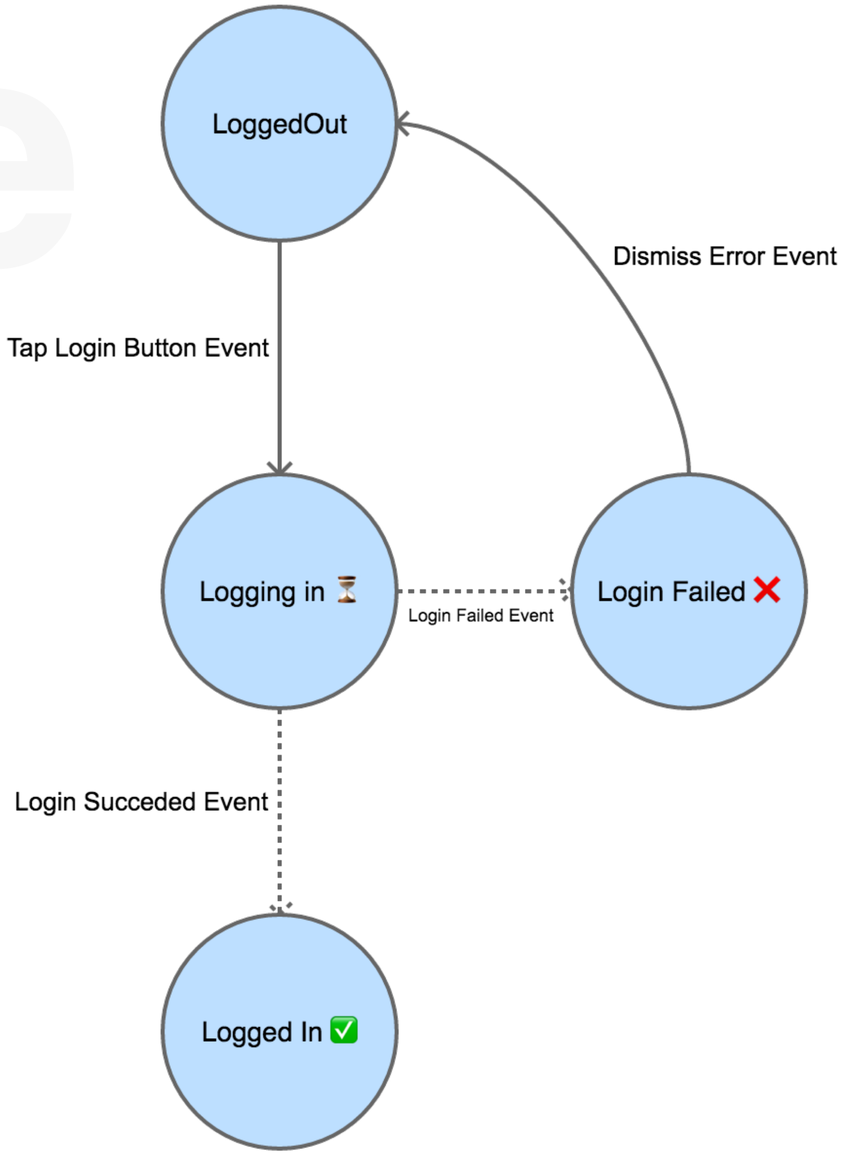
The state is stored in a Struct or an Enum

Referential Transparency

Our Reducer is a pure function which given the same input and the current state produces the same output

Sorry not very cute. Time was running out.

State graph



View Model Interface

Code finally...

```
import Foundation
import RxFeedback
import RxSwift
import RxCocoa

internal struct LoginStore {

    init(inputs: LoginView.Outputs)

    lazy var output: Driver<LoginView.Model> { mutating get set }

    enum State : Hashable {

        case loggedOut, loggedIn(User), loginFailed(APIError), performingLogin
    }
}
```

View Model Interface (Extended)

```
internal struct LoginStore {

    internal init(inputs: LoginView.Outputs)

    lazy internal var output: Driver<LoginView.Model> { mutating get set }

    /// The actions that can be performed on the view model
    internal enum Event : Equatable {
        case usernameChanged(String)
        case passwordChanged(String)
        case loginButtonTapped
        case passwordToggled
        case errorMessageDismissed
        case loginRequestSucceeded(User)
        case loginRequestFailed(APIError)
    }

    internal struct Credentials : Hashable {
        internal let username: String
        internal let password: String
    }

    internal enum State : Hashable {
        case loggedOut, loggedIn(User), loginFailed(APIError), performingLogin
    }

    internal struct StateModel : ReducibleStateWithEffects {
        internal let credentials: Credentials
        internal let isPasswordHidden: Bool
        internal let state: State
        fileprivate internal func reduce(event: Event) -> (state: StateModel, effects: Set<Effect>)
    }
}
```

State Protocol

Code finally...

```
public protocol ReducibleState: Equatable {
        associatedtype Event
        func reduce(event: Event) -> Self
    }
}
```

State Model


```
struct StateModel: ReducibleState {
    let credentials: Credentials
    let isPasswordHidden: Bool
    let state: State

    func reduce(event: Event) -> StateModel {
        switch (state, event) {
        case (.loggedOut, .togglePassword):
            return StateModel(credentials: credentials,
                              isPasswordHidden: !isPasswordHidden,
                              state: .loggedOut)

        /// …

        default:
            return (self, [])

        }
}
```

Output Driver Implementation

Code finally...

```
lazy var output: Driver<LoginView.Model> = {
    let initalStateModel = StateModel(
      credentials: Credentials(username: "", password: ""),
      isPasswordHidden: true,
      state: .loggedOut)

    return StateModel.outputStates(initialState: initalStateModel,
                                   inputEvents: inputEvents)
        .map { stateModel in LoginView.Model(stateModel: stateModel) }
        .distinctUntilChanged()
}()
```

Contains the business logic ✅

Knows the current state of the app ✅

Is easily testable

Consumes the user input ✅

Store

View

Is the view to the world ✅

Represents the current state of the app

Should not contain any logic

Passes user input to view model ✅

Questions

3

How do we test the business logic?

Rx Test

Is the state the one I expect?

Mock Input from UI Interface

Unit Test

Code finally...

```
func testIncorrectLogin() {
    let container = Container()
    let apiMock = LoginAPIMock(result: APIResult.error(APIError.wrongCredentials))

    container
        .register(LoginAPIProtocol.self) { _ in apiMock }
        .inObjectScope(.container)

    DependencyRetriever.container = container
  …
    var loginViewModel = LoginViewModel(inputs: LoginView.Outputs(usernameField: usernameSignal,
                                                                  passwordField: passwordSignal,
                                                                  loginButton: buttonClickSignal,
                                                                  registerButton: .empty(),
                                                                  showPasswordButton: .empty()),
                                                                 alertInput:.empty())
    let recorded = scheduler.record(source: loginViewModel.output)
  …
    let expectedEvents = scheduler.parseEventsAndTimes(timeline: "a-cd--e", values: expectedEventModels)

    scheduler.start()

    XCTAssertEqual(recorded.events, expectedEvents)
}
```

Contains the business logic ✅

Knows the current state of the app ✅

Is easily testable ✅

Consumes the user input ✅

Store

View

Is the view to the world ✅

Represents the current state of the app ✅

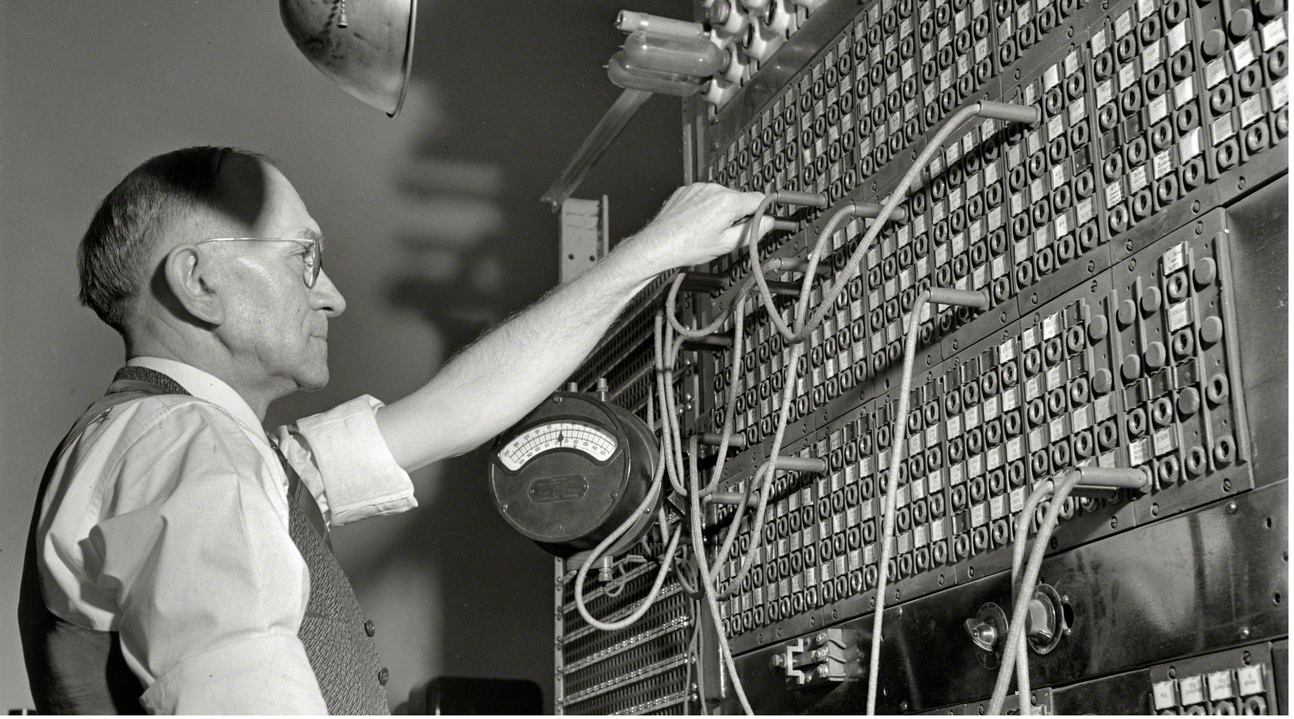
Should not contain any logic ✅

Passes user input to view model ✅

Questions

4

How do we make sure the view represents the state without the need to test it?



Snapshot Tests

One test can confirm a whole state

It is made super quickly

View Logic

Logic here is not business logic

A mistake here is only a visual glitch

The “logic” here should not be unit tested

Snaphot Test

Code finally...

```
func testSnapshot_LoggedOut_PasswordNotHidden_LoginButtonEnabled() {
    let loginViewModel = LoginView.Model(
    isLoginButtonEnabled: true,
    isPasswordHidden: false,
    isSpinning: false,
    state: .loggedOut)

    let inputs: Binder<LoginView.Model> = loginView.rx.inputs
    _ = Driver.just(loginViewModel).drive(inputs)

    recordOrAssertSnapshot(on: loginView)
}
```

Questions

5

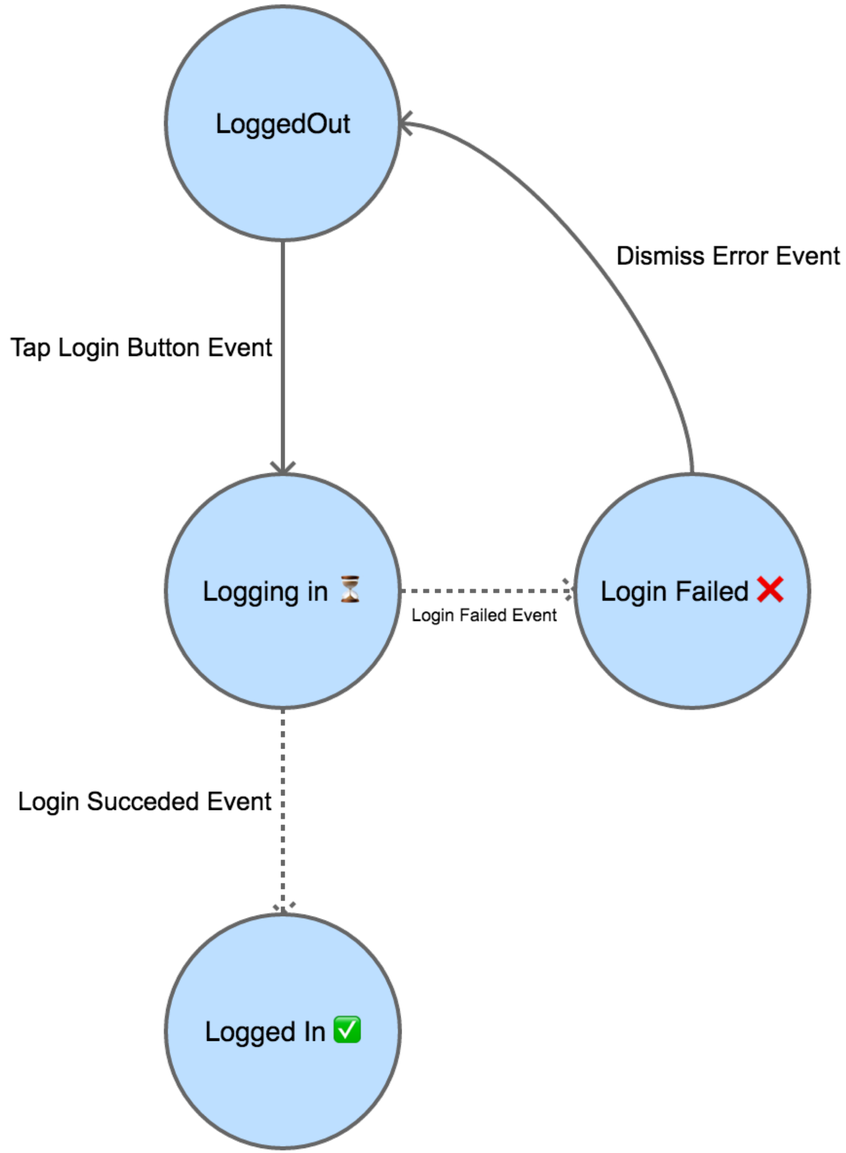
How do we handle side effects of States?

Sorry not very cute. Time was running out.

State graph

Dotted lines are side effects which produce a new state

How do we handle these side effects?



This feedback stuff is the missing puzzle piece

Get back to us (with a new State)

Feedback

RxFeedback

Elm like Commands

Reactive Automaton

Roll our own feedback mechanism

Extended State Protocol

Code finally...

```
public protocol ReducibleStateWithEffects: Hashable {
    associatedtype Effect: TriggerableEffect
    typealias Event = Effect.Event

    func reduce(event: Event) -> (state: Self, effects: Set<Effect>)
}
```

Effects Protocol

Code finally...

```
public protocol TriggerableEffect: Hashable {
    associatedtype Event

    func trigger() -> Signal<Event>
}
```

View Model Overview

Code finally...

```
internal struct LoginViewModel {

    internal enum Effect : TriggerableEffect {
        case loginRequest(loginPayload: LoginRequestModel)
        internal func trigger() -> Signal<Event>
    }

}
```

State Model

Code finally...

```
struct StateModel: ReducibleStateWithEffects {
    let credentials: Credentials
    let isPasswordHidden: Bool
    let state: State

    func reduce(event: Event) -> (state: StateModel, effects: Set<Effect>) {
        switch (state, event) {

        case (.loggedOut, .loginButtonTapped):
             return (StateModel(credentials: credentials,
                                isPasswordHidden: isPasswordHidden,
                                state: .performingLogin),
                        [.loginRequest(loginPayload: LoginRequestModel( username: credentials.username,
                                                                        password: credentials.password))])

        /// …

        default:
            return (self, [])

        }
}
```

Extension on ReducibleState

Code finally...

```
extension ReducibleStateWithEffects {

    private typealias Feedback<S, M> = (Driver<S>) -> Signal<M>

    static func outputStates(initialState: Self, inputEvents: Signal<Event>, context: Context) -> Driver<Self> {

        let inputFeedback: Feedback<StateAndEffects<Self>, Event> = { _ in return Signal<Event>.merge(inputEvents) }
        let reactFeedback: Feedback<StateAndEffects<Self>, Event> =
            react(query: { (stateAndEffects: StateAndEffects<Self>) -> Set<StateAndEffect<Self>> in
                let mappedStates = stateAndEffects.effects.map { (effect: Effect) -> StateAndEffect<Self> in
                    return StateAndEffect<Self>(state: stateAndEffects.state, effect: effect)
                }
                return Set(mappedStates)
            }, effects: { (stateAndEffect: StateAndEffect<Self>) -> Signal<Event> in
                return stateAndEffect.effect.trigger(context: context)
            })
        let reduce: (StateAndEffects<Self>, Event) -> StateAndEffects<Self> = { state, event in
            let reduced = state.state.reduce(event: event)
            return StateAndEffects<Self>(state: reduced.state, effects: reduced.effects)
        }
        let initial = StateAndEffects<Self>(state: initialState, effects: [])

        return Driver.system(initialState: initial, reduce: reduce, feedback: inputFeedback, reactFeedback)
            .map { $0.state }
    }

}
```

Questions

DEMO

Questions

?

Thanks!