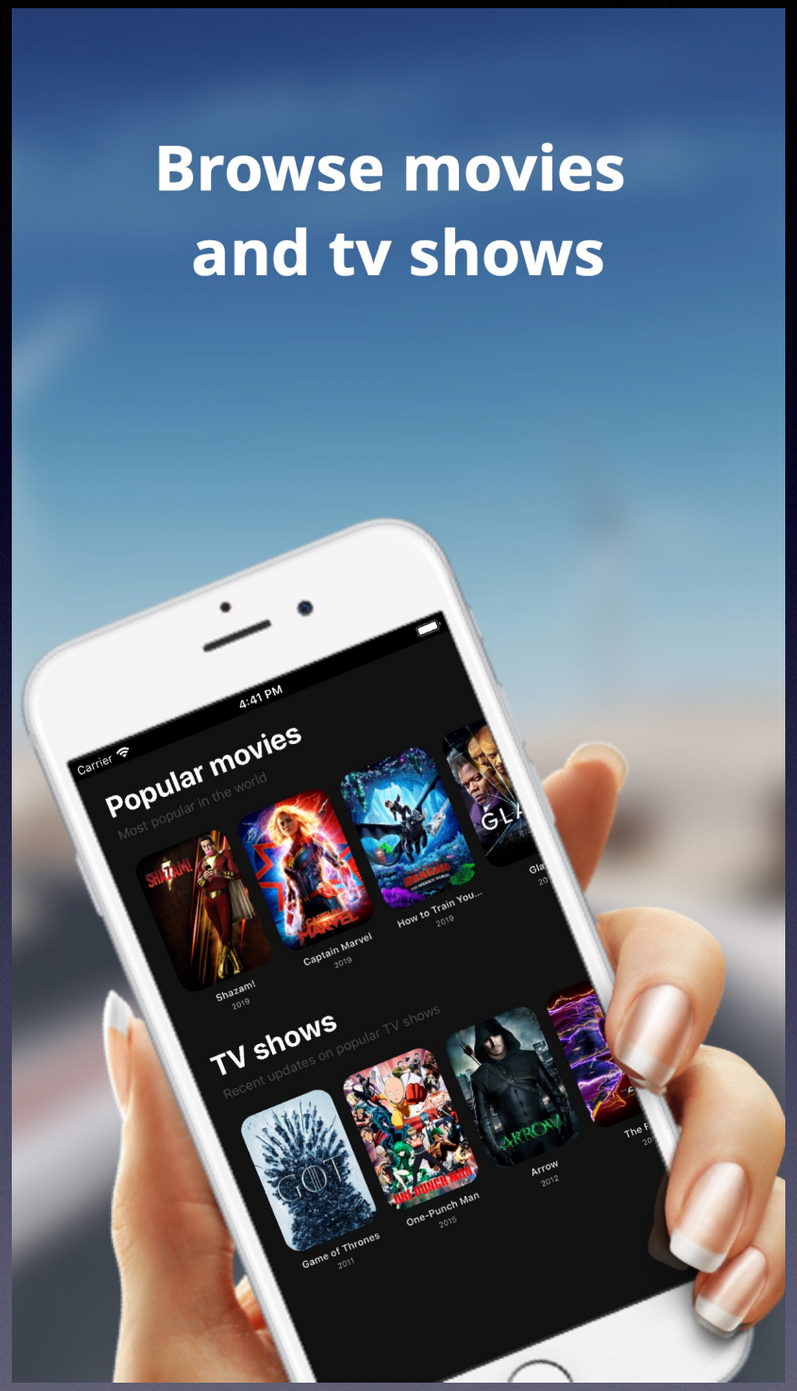
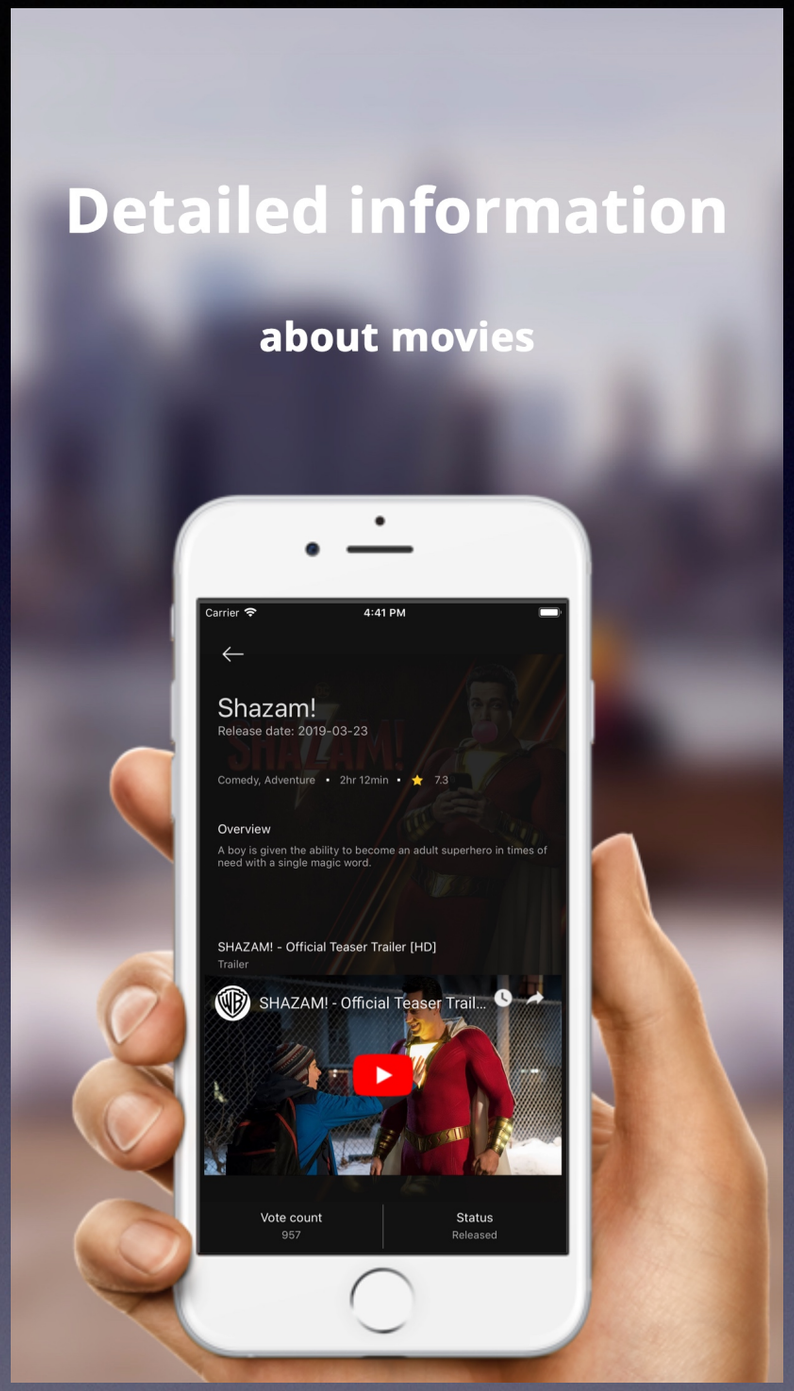
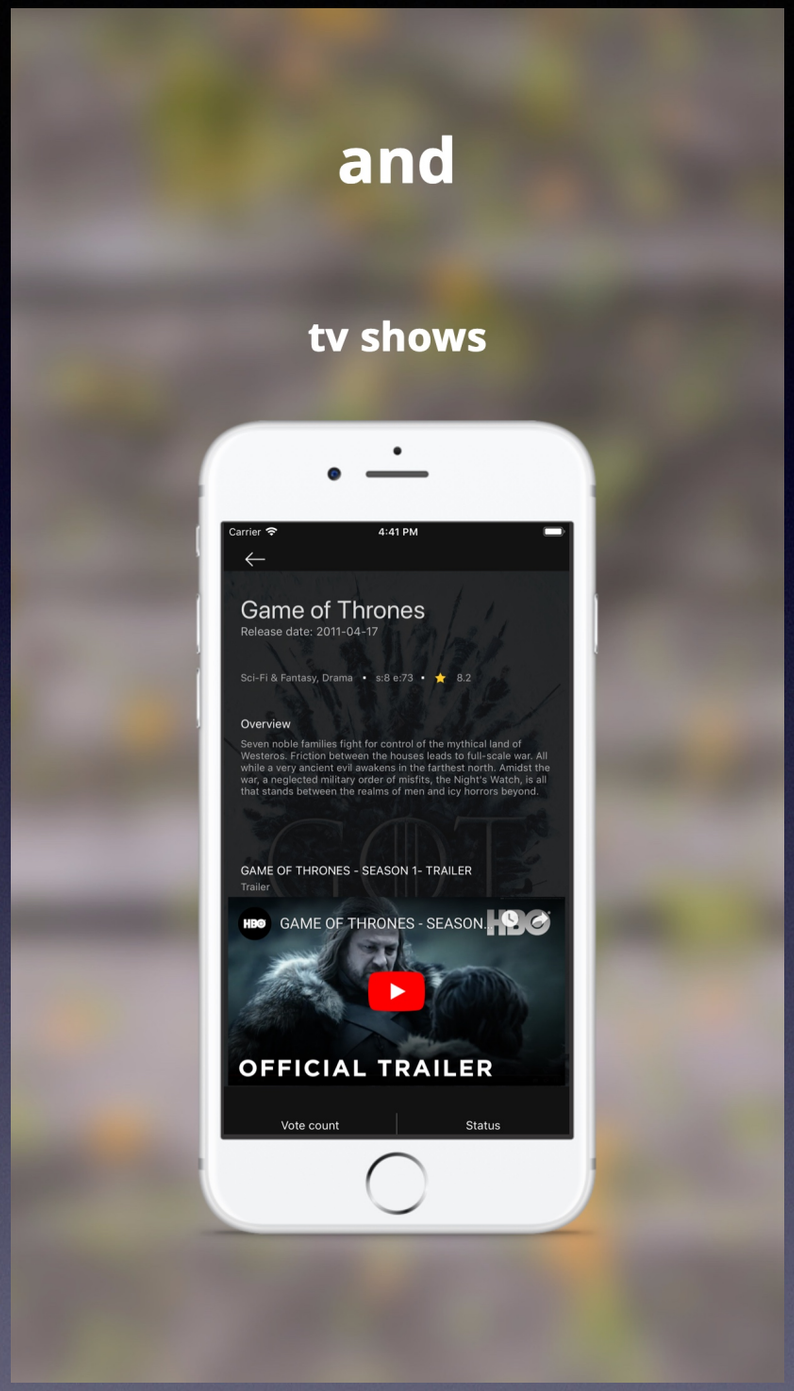
# MVVM-S-C
# Architecture in details

# Sample App

- Movie Tracker - trailers is an iOS app that connects to themoviedb.org for Movies, TV Shows, and People data. It plays movie trailers from YouTube.
- GitHub: Movie Tracker
- https://github.com/vasilevkin/MovieTracker




svasilevkin.wordpress.com

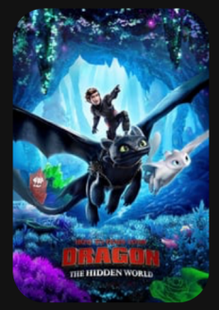
4:44

# Popular movies

Most popular in the world

Shazam!
2019

Captain Marvel
2019



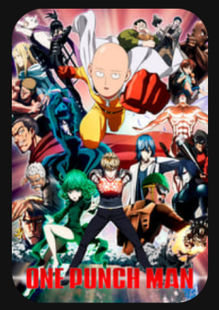

How to Train You...
2019

Gla
20

# TV shows

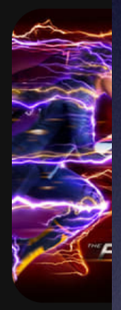
Recent updates on popular TV shows

Game of Thrones
2011




One-Punch Man
2015

Arrow
2012



The F
20

4:45

# Shazam!

Release date: 2019-03-23

Comedy, Adventure ▪ 2hr 12min ▪ ★ 7.3

## Overview

A boy is given the ability to become an adult superhero in times need with a single magic word.

SHAZAM! - Official Teaser Trailer [HD]

Trailer

| Vote count | Status |
|---|---|
| 957 | Released |

4:45

# Game of Thrones

Release date: 2011-04-17

Sci-Fi & Fantasy, Drama ▪ s:8 e:73 ▪ ★ 8.2

## Overview

Seven noble families fight for control of the mythical land of Westeros. Friction between the houses leads to full-scale war. All while a very ancient evil awakens in the farthest north. Amidst the war, a neglected military order of misfits, the Night's Watch, is all that stands between the realms of men and icy horrors beyond.

GAME OF THRONES - SEASON 1- TRAILER

Trailer

| Vote count | Status |
|---|---|
| 5574 | Returning Series |

svasilevkin.wordpress.com

# MVVM-S-C Architecture

- Consists of:
- **Model**,
- **View**,
- **ViewModel**,
- **Service**,
- **Coordinator**

# MVVM Architecture

- Original **Model-View-ViewModel** - aka **Model–View–Binder** - abstracts a view's state and behavior, creates a view model, in a manner not dependent on a specific user-interface platform.

- **MVVM** was invented by Microsoft architects to simplify event-driven programming of user interfaces for Windows Presentation Foundation (Microsoft's .NET graphics system) and Silverlight (WPF's Internet application derivative).

- GUI architecture

# Service & Coordinator

- **Model-View-ViewModel-Service-Coordinator** - adds
- **Coordinator** to separate presentation and business logic and handle routing inside application and
- **Service** for across some scenes functionality.
- MVC-S-C is a powerful approach for modern mobile applications.

# Service & Coordinator

- What are these additions:
- **Service** - separate functionality used on different scenes in app.
- **Coordinator** - separate business logic.

# Coordinator

- **Coordinator** is the most important part of this architecture.
- Implement **Coordinator** first and this greatly improve overall application structure.

# Coordinator

- Heavy usage of **Protocols** (**Interfaces** in Java).
- Everything is described as protocol firstly.
- And implemented secondly.

# Coordinator

- **Coordinator** (**Flow Coordinator**, **Navigator**) - a way to handle the app routing using some MV(X) architectures,
- it works very similar to the **routing** in VIPER pattern and is very useful if combined with MVVM and avoid a lot of boilerplate code of VIPER.

# Coordinator. Why?

- **Coordinator**, flow coordinator, navigator - separates business and presentation logic.
- A **Coordinator** is an object (Class type in Swift) which has the only one and single responsibility: to coordinate navigation from scene to scene inside application.
- What screen is presented now, what screen should be or can be shown next, can user go back to previous screen, can user go forward to the next screen, or next possible screens.
- It's pure business logic. Coordinator knows nothing about ui and details of presentation, only sequence of screens and transitions.

# Coordinator responsibilities

- Instantiate ViewController and related ViewModel
- Instantiate required Services
- Instantiate and inject Dependencies into the ViewController and ViewModel
- Present, show or push ViewController to the screen

# Coordinator advantages

- Move navigation logic from **ViewController** to **Coordinator**. So **ViewController** implements single responsibility principle. **ViewController** can be a simple wrapper around its views and subviews presented on the screen and it handles only UIKit related presentation logic.
- Each **ViewController** doesn't know what **ViewController** came before it, which one come next, or what dependencies they relay and pass on to the next **ViewController**.
- Each **ViewController** can be instantiated individually and separately from other **ViewControllers**.
- **ViewController** doesn't know how data is processed and doesn't know application structure.

# Coordinator types

- App contains two main coordinator types:
- **Coordinator** - for all each separate scene or for related sequence of scenes.
- **AppCoordinator** - for entire app.

# AppCoordinator

- Main in the **Coordinators** hierarchy.
- Only one for the whole app.
- Main entry point, owned by AppDelegate.
- Never leaves stack, always has strong pointer from AppDelegate.

# Coordinators structure

- App consists of multiple coordinators,
- one for each flow or scene.
- All other coordinators in the application are children of main **AppCoordinator**.

# Coordinator parts

- Coordinator protocol
- Additions for Coordinator protocol
- CoordinatorChild enum
- CoordinatorDelegate protocol

# Coordinator protocol

- Each coordinator must implement these methods and variables.

```
/**
 Base Abstract Coordinator for all coordinators on concrete scenes.
 Based on UINavigationController.
 */
protocol Coordinatable {
    /**
     Navigation controller for a current coordinator.
     A container view controller that defines a stack-based scheme
         for navigating hierarchical content.
     */
    var navigationController: UINavigationController { get }
    /**
     Required init
     - Parameter navigationController: container view controller
     */
    init(navigationController: UINavigationController)
    /**
     An entry point for coordinator to begin work and navigation.
     */
    func start()
    /**
     Transition back to the presenting View Controller
     */
    func goBack()
}
```

# Additions for Coordinator protocol

- In addition some concrete coordinator for specific screen flow can implement other methods,
- for example:

```
/**
 Coordinator for all transitions on Featured scene
 */
protocol FeaturedCoordinatable: Coordinatable {

    /**
     Transition to Movie detail scene
     - Parameter movieId: unique movie identifier (id)
     - Parameter api: service to provide all required detail data
     */
    func navigateToMovieDetailScene(with movieId: Int, api:
        ApiThemoviedbService)

    /**
     Transition to TV Show detail scene
     - Parameter tvShowId: unique tv show identifier (id)
     - Parameter api: service to provide all required detail data
     */
    func navigateToTVShowDetailScene(with tvShowId: Int, api:
        ApiThemoviedbService)

}
```

```swift
/**
 Possible child coordinators for Coordinator.
 */
enum CoordinatorChild: Hashable {

}


/**
 Possible child coordinators for FeaturedCoordinator.
 */
enum FeaturedCoordinatorChild: Hashable {
    case movieDetail
    case tvShowDetail
}
```

CoordinatorChild enum

```swift
/**
 Parent delegate protocol for Coordinator.
 Should be implemented by Parent Coordinator who invoked Coordinator.
 */
protocol CoordinatorDelegate: AnyObject {

    /**
     Invoked when the Coordinator is no longer needed
     Coordinator never finishes by itself,
     but calls its Parent Coordinator only when it finished work.
     */
    func coordinatorDidFinish()

}
```

CoordinatorDelegate protocol

```swift
/**
 Parent delegate protocol for FeaturedCoordinator.
 Should be implemented by parent Coordinator who invoked
     FeaturedCoordinator.
 */
protocol FeaturedCoordinatorDelegate: AnyObject {

    /**
     Invoked when the featured flow is no longer needed
     Coordinator never finishes by itself,
     but calls its Parent Coordinator only when it finished work.
     */
    func featuredCoordinatorDidFinish()

}
```

Example implementation

```
extension ApplicationCoordinator: FeaturedCoordinatorDelegate {

    /**
     Invoked when the featured flow is no longer needed.
     */
    func featuredCoordinatorDidFinish() {
        dLog("ApplicationCoordinator:: childCoordinators[.featured] =
            nil")

        childCoordinators[.featured] = nil
        navigationController.popViewController(animated: true)

        dLog("ApplicationCoordinator:: childCoordinators =
            \(childCoordinators)")

        showLoginScene()
    }
}
```

Method implementation

# Model

- **Model** refers either to
- a **domain model**, which represents real state content (an object-oriented approach), or to
- the **data access layer**, which represents content (a data-centric approach).

```
/**
 Data struct model for Movie
 Detail description:
 https://developers.themoviedb.org/3/movies/get-movie-details
 */
struct Movie: Decodable {
    let budget: Int?
    let genres: [Genre]?
    let homepage: String?
    let movieId: Int
    let imdbId: String?
    let title: String

    enum CodingKeys: String, CodingKey {
        case budget = "budget"
        case genres =  "genres"
        case homepage = "homepage"
        case movieId = "id"
        case imdbId = "imdb_id"
        case title = "title"
    }
}
```

# View

- **View** is a structure, layout, and appearance of what a user sees on the screen.
- Displays a representation of the model and receives the user's interaction with the view (clicks, gestures), and it forwards the handling of these to the **ViewModel** via the data binding (properties, event callbacks).

# View

- All Views are implemented as separate reusable views in .xibs. General to use on different scenes.
- Storyboards contain only skeleton for each scene and does not contain any hardcoded segues between scenes.

# View

- All Outlets for each individual view are marked as private. And can only be modified via view's public configure method.

- Visual parameters of each view (fonts, colors, sizes) are configured inside view's setup method, which is called on view instantiation.

- There are no any "magic numbers" in implementation of each view, but only Constants, stored as global static constants, unified for entire app.

```
class ItemDetailVideoView: UIView {

    @IBOutlet private weak var trailerTitleLabel: UILabel!
    @IBOutlet private weak var trailerSubtitleLabel: UILabel!
    @IBOutlet private weak var videoPlayerView: VideoPlayerView!
    @IBOutlet private var contentView: UIView!

    override init(frame: CGRect) {
        super.init(frame: frame)
        setup()
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        setup()
    }
}
```

```
func configure(with title: String,
               subtitle: String,
               videoPlayerId: String) {
    self.trailerTitleLabel.text = title
    self.trailerSubtitleLabel.text = subtitle

    dLog("play Youtube video with id = \(videoPlayerId)")
    videoPlayerView.playYoutube(with: videoPlayerId)
}

private func setup() {
    Bundle.main.loadNibNamed("ItemDetailVideoView",
                             owner: self, options: nil)
    addSubview(contentView)

    self.trailerTitleLabel.textColor    =
        Constants.uiSubviewTitleTextColor
    self.trailerSubtitleLabel.textColor =
        Constants.uiSubviewSubtitleTextColor

    self.trailerTitleLabel.font    = Constants.uiSubviewTitleFont
    self.trailerSubtitleLabel.font =
        Constants.uiSubviewSubtitleFont
}
```

```swift
extension ItemDetailVideoView {

    func configure(with data: AdditionalMediaItemDetailData) {
        configure(with: data.videoName ?? "Video unavailable",
                  subtitle: data.videoType ?? "n/a",
                  videoPlayerId: data.videoKey ?? "")
    }

}
```

# ViewModel

- **ViewModel** is an abstraction of the **View** exposing public properties and commands.
- Instead of the controller of the MVC pattern, or the presenter of the MVP pattern, MVVM has a **Binder**, which automates communication between the view and it's bound properties in the ViewModel.

# ViewModel

- **Bindings** can be implemented using different approaches, such as RxSwift, Reactive Cocoa, KVO, didSet, willSet, observers, promises.

- **Bindings** used to simplify data flows and state management.

- **RxSwift** is a production mature and developed framework for declarative programming paradigm with a great management of ***asynchronous callbacks*** as well as different edge cases for ***state***.

- **RxSwift** is not a magic but ***asynchronous code*** and ***data flows***.

# ViewModel

- **ViewModel** is a state of the data in the model.
- The main difference between the ViewModel and the Presenter in the MVP pattern, is that the presenter has a reference to a view whereas ViewModel does not.
- Instead, a View directly binds to properties on ViewModel to send and receive updates.
- To function efficiently, this requires a binding technology or generating boilerplate code to do the binding.

# ViewModel responsibilities

- ViewModel takes Input parameters,
- requests data from Dependencies services
- and transform result data to appropriate structure described in Output parameters.

# ViewModelType protocol

```
/**
 Type to describe view model
 Consists of:
 Input: struct describing input parameters
 Output: struct describing output parameters
 Dependencies: services required for correct
               data processing inside a view model
 Transform: transforming method
 */
protocol ViewModelType {
    associatedtype Input
    associatedtype Output
    /**
     Services required for correct data processing inside a view model
     */
    associatedtype Dependencies

    /**
     Get input and transforms it to output specific for view
         controller
     - Parameter input: struct describing input parameters
     - Returns: struct describing output parameters
     */
    func transform(input: Input) -> Output
}
```

# Example of ViewModel

```
final class ItemDetailViewModel: ViewModelType {
    struct Input {
        let ready: Driver<Void>
        let backTrigger: Driver<Void>
    }
    struct Output {
        let data: Driver<ItemDetailData?>
        let mediaData: Driver<AdditionalMediaItemDetailData?>
        let back: Driver<Void>
    }
    struct Dependencies {
        let itemDetailType: ItemDetailType
        let itemId: Int
        let api: ApiThemoviedb
        let coordinator: ItemDetailCoordinatable
    }
    private let dependencies: Dependencies

    init(dependencies: Dependencies) {
        self.dependencies = dependencies
    }
    func transform(input: ItemDetailViewModel.Input) ->
        ItemDetailViewModel.Output {
        // ... concrete transform implementation ...
    }
}
```

# ViewModel Binder

- **Binder** - declarative data.
- Binder frees the developer from writing boiler-plate logic to synchronize ViewModel and View via ViewController.

# Example of Bindings in ViewController

```swift
class ItemDetailViewController: UIViewController {
    @IBOutlet private weak var headerView: ItemDetailHeaderView!
    @IBOutlet private weak var videoView: ItemDetailVideoView!
    @IBOutlet private weak var tipsView: ItemDetailTipsView!
    @IBOutlet private weak var posterImageView: GradientImageView!
    @IBOutlet private weak var backButton: UIButton!

    var viewModel: ItemDetailViewModel?
    private let disposeBag = DisposeBag()

    override func viewDidLoad() {
        super.viewDidLoad()
        bindViewModel()
    }
```

All Outlets are declared as private to protect from external modification.

```
private func bindViewModel() {
    let input = ItemDetailViewModel.Input(ready:
        rx.viewWillAppear.asDriver(),
                                          backTrigger:
                                          backButton.rx.tap
                                          .asDriver())

    let output = viewModel?.transform(input: input)
```

```
private func bindViewModel() {
    let input = ItemDetailViewModel.Input(ready:
        rx.viewWillAppear.asDriver(),
                                          backTrigger:
                                           backButton.rx.tap
                                           .asDriver())

    let output = viewModel?.transform(input: input)
```

All data processing implemented in ViewModel.

```
    output?.mediaData
        .drive(onNext: { [weak self] mediaData in
            guard let mediaData = mediaData,
                let strongSelf = self else {
                    dLog("Unexpectedly found nil")
                    return
            }
            strongSelf.videoView.configure(with: mediaData)
        })
        .disposed(by: disposeBag)

    output?.back
        .drive()
        .disposed(by: disposeBag)
}
```

Each reusable View instantiated and configured with data from ViewModel Output.

```
output?.data
    .drive(onNext: { [weak self] data in
        guard let data = data,
            let strongSelf = self else {
                dLog("Unexpectedly found nil")
                return
        }
        strongSelf.headerView.configure(with: data)
        strongSelf.tipsView.configure(with: data)

        if let urlString = data.posterPath, let url =
            URL(string: urlString) {
            // Load image using Nuke
            Nuke.loadImage(with: url, into:
                strongSelf.posterImageView)
        }
    })
    .disposed(by: disposeBag)
```

# Service

- **Service** is usually a stateless object responsible for some small piece of functionality,
- for example: network requests or location.

```swift
/**
 Service communicates with themoviedb.org api
 Consists of separate providers
 */
protocol ApiThemoviedbService: ApiThemoviedbMoviesProvider,
                               ApiThemoviedbTVShowsProvider,
                               ApiThemoviedbAuthProvider {

}
```

ApiThemoviedbService consists of separate Service Providers.

svasilevkin.wordpress.com

```swift
/**
 Gets Movie data from themoviedb.org api
 */
protocol ApiThemoviedbMoviesProvider {
    /**
     Fetches popular movies from themoviedb.org api
     - Returns: Optional observable array of 'Movie' type
     */
    func fetchPopularMovies() -> Observable<[Movie]?>

    /**
     Fetches movie details for single movieId from themoviedb.org api
     - Parameter movieId: unique movie identifier (id)
     - Returns: Optional observable of 'Movie' type
     */
}
```

Protocol to describe MoviesProvider

```swift
protocol HTTPClientService {
    /**
     Sends a 'get' request to a specified URL
     - Parameter url: String of URL
     - Returns: Optional observable Data type
     */
    func get(url: String) -> Observable<Data?>

    /**
     Sends a 'post' request to a specified URL
     - Parameter url: String of URL
     - Parameter params: Json object data
     - Returns: Optional observable response Data type
     */
    func post(url: String, params: [String: Any]) -> Observable<Data?>
}
```

Protocol to describe HTTPClientService, used to communicate with server via network

# Conclusion

- Model-View-ViewModel-Service-Coordinator - is a further development of MVVM.

# MVVM-S-C advantages

- Separate business and presentation logic
- Clear project structure
- Reusable views
- Each ViewController can be instantiated separately
- Great state management
- Reactive approach

# MVVM-S-C advantages

- Fast development time when adding new similar scenes and data structures
- Easy to change application backend server provider
- Highly testable
- Each functionality implemented only once and can be used in different scenes
- Easy to find and modify any small piece of functionality

# MVVM-S-C disadvantages

- Overkill for small and simple apps
- Required some traction for new developers
- Coordinators structure can be very big for app with really many screens
- Not cross-platform approach
- Requires additional steps for new feature implementation (implement protocols for services, services, protocols for coordinators, coordinators, protocols for viewModels, viewModels)

# Links

- My blog: https://svasilevkin.wordpress.com/
- GitHub Movie Tracker https://github.com/vasilevkin/MovieTracker
- Twitter: Sergey Vasilevkin @vasilevkin