MVVM

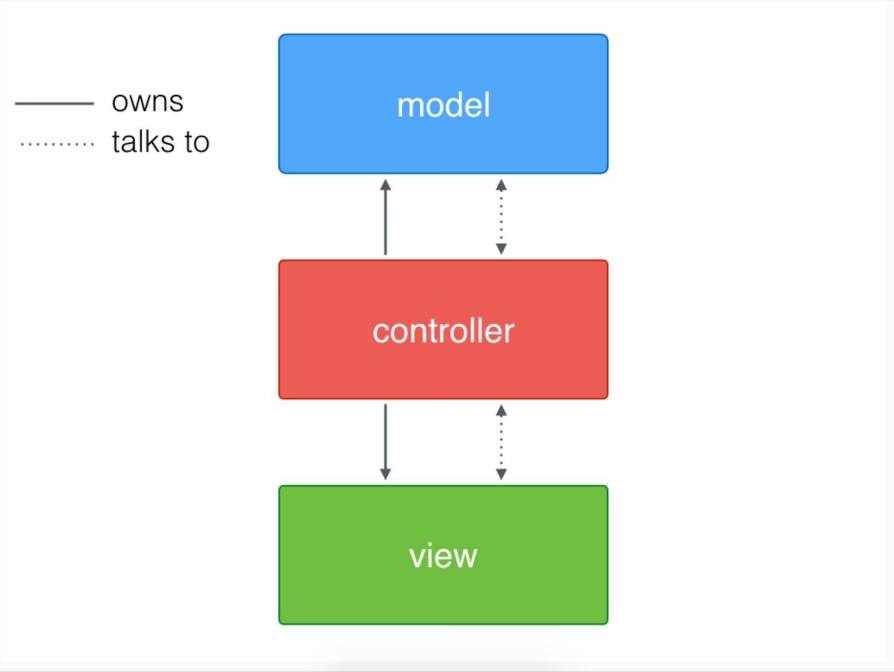
Topics

- Dark side of MVC
- MVVM
- Testing using MVVM
- Protocol oriented MVVM

MVC

What is it?

Traditional MVC



Model :

- Logic and computation
- Application state
- Reading and writing data
- May include Networking and Data Validation

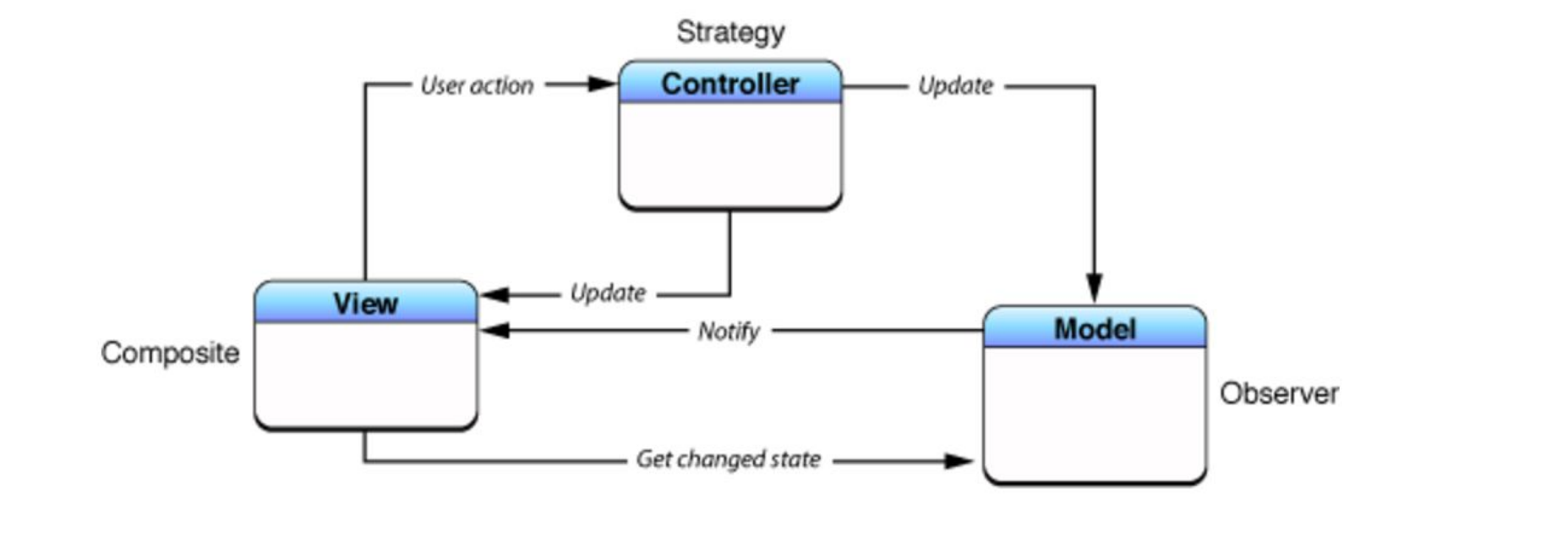
View:

- Presenting data
- Handling user interaction

Controller:

- Glue Model and views
- Modeling data

Cocoa MVC


Strategy
Controller
User action
Update
View
Update
Notify
Model
Composite
Observer
Get changed state

Disadvantages of Cocoa MVC in iOS

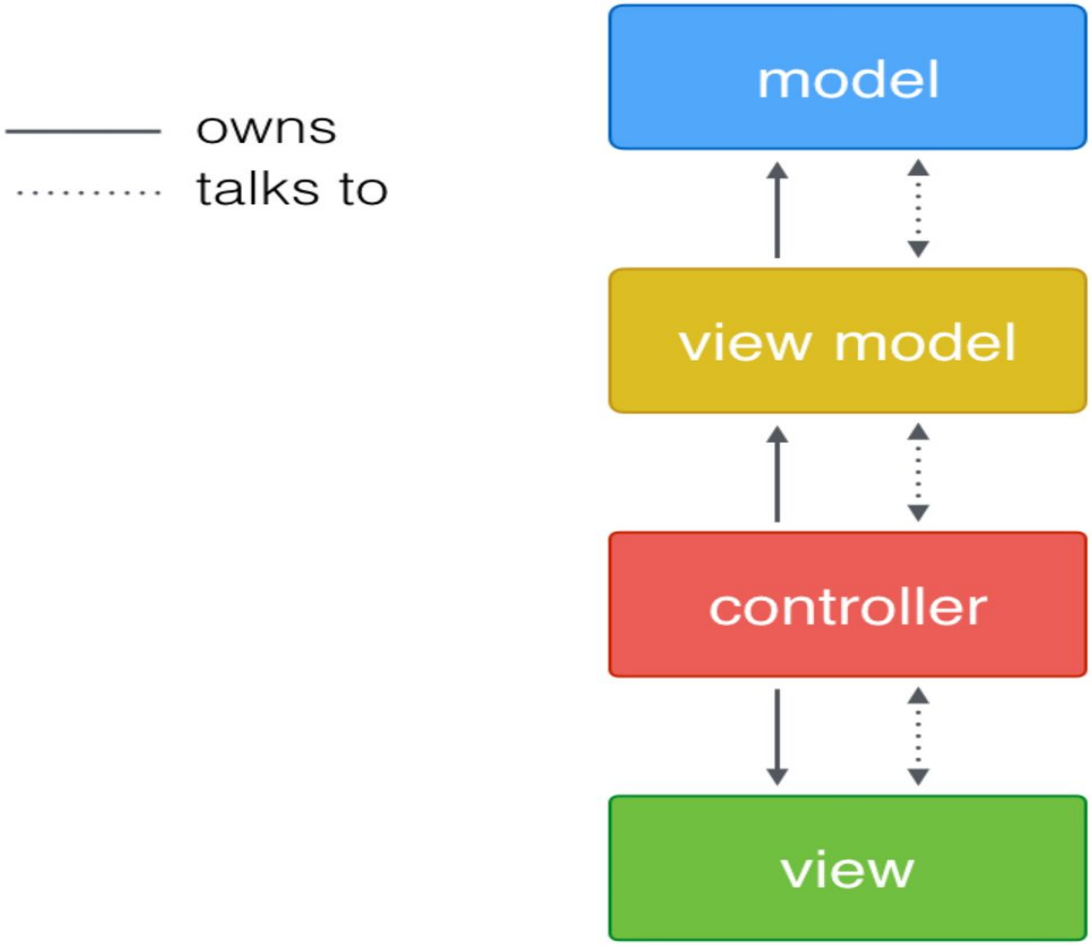
1. A lot of the code you write does not belong in the view or model layer. No problem. Dump it in the controller. Problem solved. Right? Not really.
2. Resulting in Massive View Controller. And complex view controllers
3. Computations like Data formatting and validation happen in ViewController and are quite tightly coupled to the views, making those computations and views non-reusable
4. Writing tests becomes a tedious task
5. Any change in Model affects the View Controller and the views

MVVM

What is it?

- Architectural pattern derived from MVC.
- MVVM was developed by Microsoft's architects Ken Cooper and Ted Peters in 2005
- Facilitates separation of GUI code with the Business logic(Single responsibility)

1. ViewModel => Model Value Convertor
2. Hence, ViewModel is more Model than View. Infact, no view should exist in ViewModel
3. Model need not be changed for change in Views
4. Or, views/ViewControllers need not be changed for the change in Model.


owns
talks to
model
view model
controller
view

Data model

- Refers to Real state content Or Data Access Layer(persistent DB)
- an object that has the single responsibility of retrieving data from any sort of endpoint (REST API, Core Data, file system, etc.) and making it available to other components
- Has no reference to the ViewModel Or View Controller(or views)

View

- Refers to ViewController/Views
- the View is the structure, layout, and appearance of what a user sees on the screen.
- Responsible for presenting data to the user.
- Also responsible for handling user interaction, animations, etc.
- No access to the Model
- Controller binds ViewModel and View together

ViewModel

- Black-box of data. VC assumes data is right always.
- Parent view model and child view models can exist
- When testing your View Model's data validation/rules, you can stub out the Session Manager entirely and only test the View Model

ViewModel - core principle

> The view model exposes an interface to the controller for displaying information about the model. The controller does not have direct access to the model.

Models:

```
@objc(Category) open class Category: NSManagedObject {


    @NSManaged var defaultPrice: NSObject?
    @NSManaged var userType: String?
    @NSManaged var categoryQuestionsId: String?
    @NSManaged var minimumBookingAmount: NSDecimalNumber?

    var sortedQuestions : NSOrderedSet? {

        if _sortedQuestions == nil {
            if let array =
                self.questionRelationship?.sortedArray(using:
                    [NSSortDescriptor(key: "orderIndex", ascending: true)]) {
                _sortedQuestions = NSOrderedSet(array: array)
            }
        }
        return _sortedQuestions
    }
```

```
enum SegmentType {
    case Main
    case WarmUp
    case CoolDown
}

class Segment {
    var enabled: Bool
    var type: SegmentType
    var sounds = [Sound]()
    var duration: Double = 300

    init(type: SegmentType) {
        self.type = type
        self.enabled = true
    }

    func soundOfType(type: SoundType) -> Sound? {
        var result: Sound? = nil

        for sound in sounds {
            if sound.type == type {
                result = sound
            }
        }

        return result
    }

}
```

ViewModel initialization

```
/* In Navigator/FlowController class */

class func getProfileViewController(assembly: TyphoonAssembly, profileId: String, successBlock: @escaping
    ((ProfileViewController?) -> Void), errorBlock: @escaping((Error) -> Void)) {

    if let profileVC = assembly.componentForKey(ProfileViewController.assemblyName) as? ProfileViewController {

        /* This could be a local database call too */

        APIHandler.getProfile(forId id: String, successBlock: { (profile) in


            profileVC.profileViewModel = ProfileViewModel(withProfile: profile)
            return profileVC


        }, errorBlock: { (error) in
            errorBlock(error)
        })


    }
    return nil
}
```

ViewModel

```
public class ListViewModel {

    public let context: Context

    public init(_ context: Context) {
        self.context = context
    }

    public weak var delegate: ListViewModelDelegate?

    public func handleAddPressed() {
        delegate?.showAddView()
    }

    public struct Item {
        public let title: String
        public let subtitle: String
        public let amount: String
    }

}

public protocol ListViewModelDelegate: class {

    func showAddView()

}
```

Let’s look at a simple project to understand ViewModels better…

Unit testing

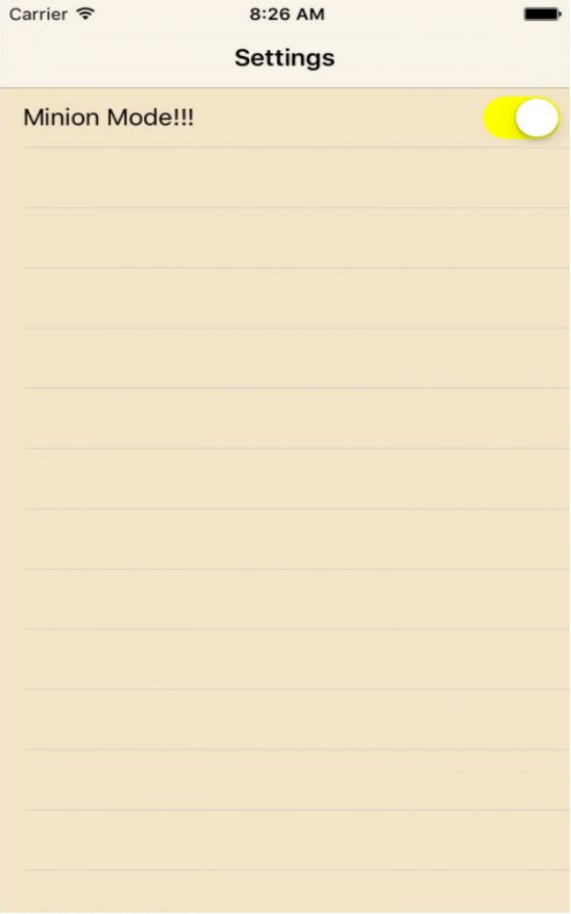
View controllers are notoriously hard to test. Ironically, the Model-View-Controller pattern forces developers to put a lot of the heart and brains of their applications in view controllers.

The Model-View-ViewModel pattern makes testing much easier, another key feature of MVVM.

Protocol Oriented MVVM

An interesting and much cleaner approach to using MVVM is via Protocols.

Consider a simple tableview and a single UITableViewCell with UILabel and a UISwitch



Our TableViewCell might just look like this. And similarly, it would look unclean in the cellForRowAtIndexPath too.

```
class SwitchWithTextTableViewCell: UITableViewCell {


    func configure(title: String, titleFont: UIFont,
                   titleColor: UIColor,
                   switchOn: Bool,
                   switchColor: UIColor = UIColor.green,
                   onSwitchToggleHandler: onSwitchToggleHandlerType? = nil)
    {

        /* Configure the view here */

    }
}
```

- Two protocols and a protocol extension which can by any other UILabel and UISwitch
- Now, our SwitchWithTextTableViewCell confirms to SwitchWithTextViewPresentable which comprises of two protocols. Now the configure method looks cleaner

```swift
// your label protocol
protocol TextPresentable {
    var text: String { get }
    var textColor: UIColor { get }
    var font: UIFont { get }
}

// your switch protocol
protocol SwitchPresentable {
    var switchOn: Bool { get }
    var switchColor: UIColor { get }
    func onSwitchTogleOn(on: Bool)
}

extension SwitchPresentable {
    var switchColor: UIColor { return .yellow }
}
```

```swift
//  SwitchWithTextTableViewCell.swift

// protocol composition
// based on the UI components in the cell
typealias SwitchWithTextViewPresentable = TextPresentable & SwitchPresentable

class SwitchWithTextTableViewCell: UITableViewCell {

    @IBOutlet private weak var label: UILabel!
    @IBOutlet private weak var switchToggle: UISwitch!

    private var delegate: SwitchWithTextViewPresentable?

    // configure with something that conforms to the composed protocol
    func configure(withPresenter presenter: SwitchWithTextViewPresentable) {
        delegate = presenter

        // configure the UI components
        label.text = presenter.text

        switchToggle.isOn = presenter.switchOn
        switchToggle.onTintColor = presenter.switchColor
    }

    @IBAction func onSwitchToggle(sender: UISwitch) {
        delegate?.onSwitchTogleOn(on: sender.isOn)
    }
}
```

ViewModel confirms to SwitchWithTextViewPresentable typealias. And implements all the properties and methods required by the two protocols

```swift
//  MyViewModel.swift

struct MinionModeViewModel: SwitchWithTextViewPresentable {
    //    This would usually be instantiated with the model
    //    to be used to derive the information below
    //    but in this case, my app is pretty static
}

// MARK: TextPresentable Conformance
extension MinionModeViewModel {
    var text: String { return "Minion Mode" }
    var textColor: UIColor { return .black }
    var font: UIFont { return .systemFont(ofSize: 17.0) }
}

// MARK: SwitchPresentable Conformance
extension MinionModeViewModel {
    var switchOn: Bool { return false }
    var switchColor: UIColor { return .yellow}

    func onSwitchTogleOn(on: Bool) {
        if on {
            print("The Minions are here to stay!")
        } else {
            print("The Minions went out to play!")
        }
    }
}
```

Mission Accomplished!!!

My View controller(cellforrowatindexpath) looks soo pretty now!

```swift
//  MyTableViewController.swift

 func tableView(tableView: UITableView,
                        cellForRowAtIndexPath indexPath: IndexPath) -> UITableViewCell
{
    let cell = tableView.dequeueReusableCell(withIdentifier: "SwitchWithTextTableViewCell", for: indexPath) as!
        SwitchWithTextTableViewCell

    // this is where the magic happens!
    let viewModel = MinionModeViewModel()
    cell.configure(withPresenter: viewModel)
    return cell
}
```

Is this pattern scalable?

```swift
//  SwitchWithTextTableViewCell.swift

// protocol composition
// based on the UI components in the cell
typealias SwitchWithTextViewPresentable = ImagePresentable & TextPresentable & SwitchPresentable

class SwitchWithTextTableViewCell: UITableViewCell {

    @IBOutlet private weak var label: UILabel!
    @IBOutlet private weak var switchToggle: UISwitch!

    // new IBOutlet for the UIImageView
    @IBOutlet private weak var iconView: UIImageView!

    private var delegate: SwitchWithTextViewPresentable?

    // configure with something that conforms to the composed protocol
    func configure(withPresenter presenter: SwitchWithTextViewPresentable) {
        delegate = presenter

        // configure the UI components
        label.text = presenter.text

        switchToggle.isOn = presenter.switchOn
        switchToggle.onTintColor = presenter.switchColor

        /* *** */
        // adding the configuration of the image
        iconView.image = UIImage(named: presenter.imageName)
    }

    @IBAction func onSwitchToggle(sender: UISwitch) {
        delegate?.onSwitchTogleOn(on: sender.isOn)
    }
}
```

```swift
//  MyViewModel.swift

struct MinionModeViewModel: SwitchWithTextViewPresentable {
    //    This would usually be instantiated with the model
    //    to be used to derive the information below
    //    but in this case, my app is pretty static
}

// MARK: TextPresentable Conformance
extension MinionModeViewModel {
    var text: String { return "Minion Mode" }
    var textColor: UIColor { return .black }
    var font: UIFont { return .systemFont(ofSize: 17.0) }
}

// MARK: SwitchPresentable Conformance
extension MinionModeViewModel {
    var switchOn: Bool { return false }
    var switchColor: UIColor { return .yellow}

    func onSwitchTogleOn(on: Bool) {
        if on {
            print("The Minions are here to stay!")
        } else {
            print("The Minions went out to play!")
        }
    }
}

/* *** */
//MARK: New protocol conformance
extension MinionModeViewModel {
    var imageName: String { return "minion.png" }
}
```

Criticism

- ‘Overkill’ for simple UI
- Generalizing the ViewModel becomes more difficult.
- High memory footprint.

Thank you