Ravi Desai

PRACTICAL ADVICE ON DESIGNING FOR TESTABILITY

# WHEN SWIFT BREAKS YOUR UNIT TESTS

# HOW THIS TALK CAME TO BE

- Had a unique opportunity to rewrite an app (start over)
  - Swift - wanted full use of value types and functional
  - Test oriented
- Tools / techniques didn't always translate
  - Some updated for Swift as time went on
  - Some couldn't be 'fixed'
  - Relied on ObjC  dynamism and reflection

# HALLMARKS OF A SUCCESSFUL SUITE OF UNIT TESTS

- Tests should be:
  - Fast: >6 seconds and I'm off checking Facebook
  - Succinct: Too long and you can't follow it
  - Robust: "Works on my machine!"
  - Indicative: On failure you know what's wrong
  - Complete: No code that is "too hard" to test

# DESIGNING FOR TESTABILITY

- Code should propagate errors - not “eat” them
- A unit of code should not try to do too much
- Asynchronous code should be isolated
- Apps should have layers

# APPLICATION LAYERS

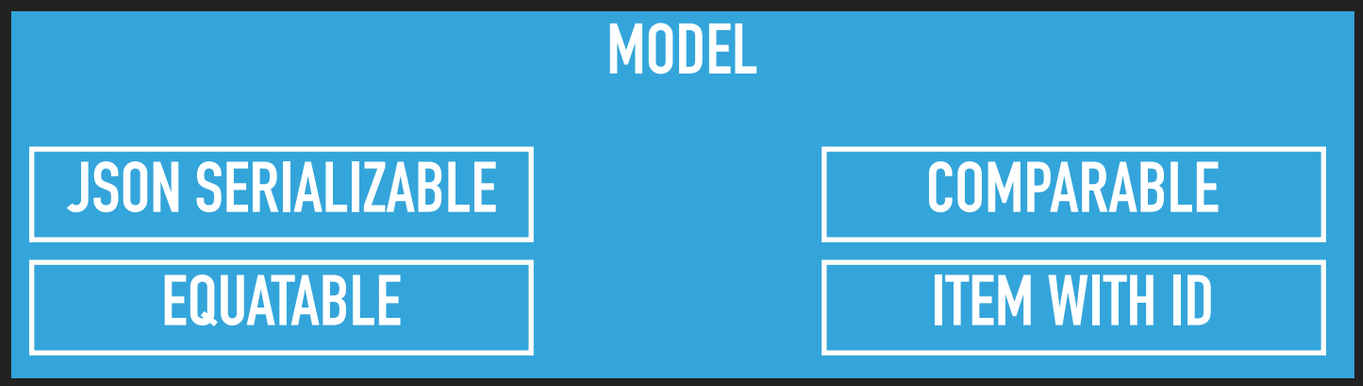
- Model Layer
- Network Services API Layer
- View Model Layer
- Controller Layer

# MY FAKE APP

- Needed some scaffolding for this talk
  - App isn't compelling in any way - it isn't even complete
  - App crams several different styles together
  - Only goal is to demonstrate testing
- But the structure of the app is useful

# MODEL LAYER

- Serialization (JSONSerializable)
- Equatable (operator ==)
- Comparable (operator <)
- Unique ID (NSUUID)

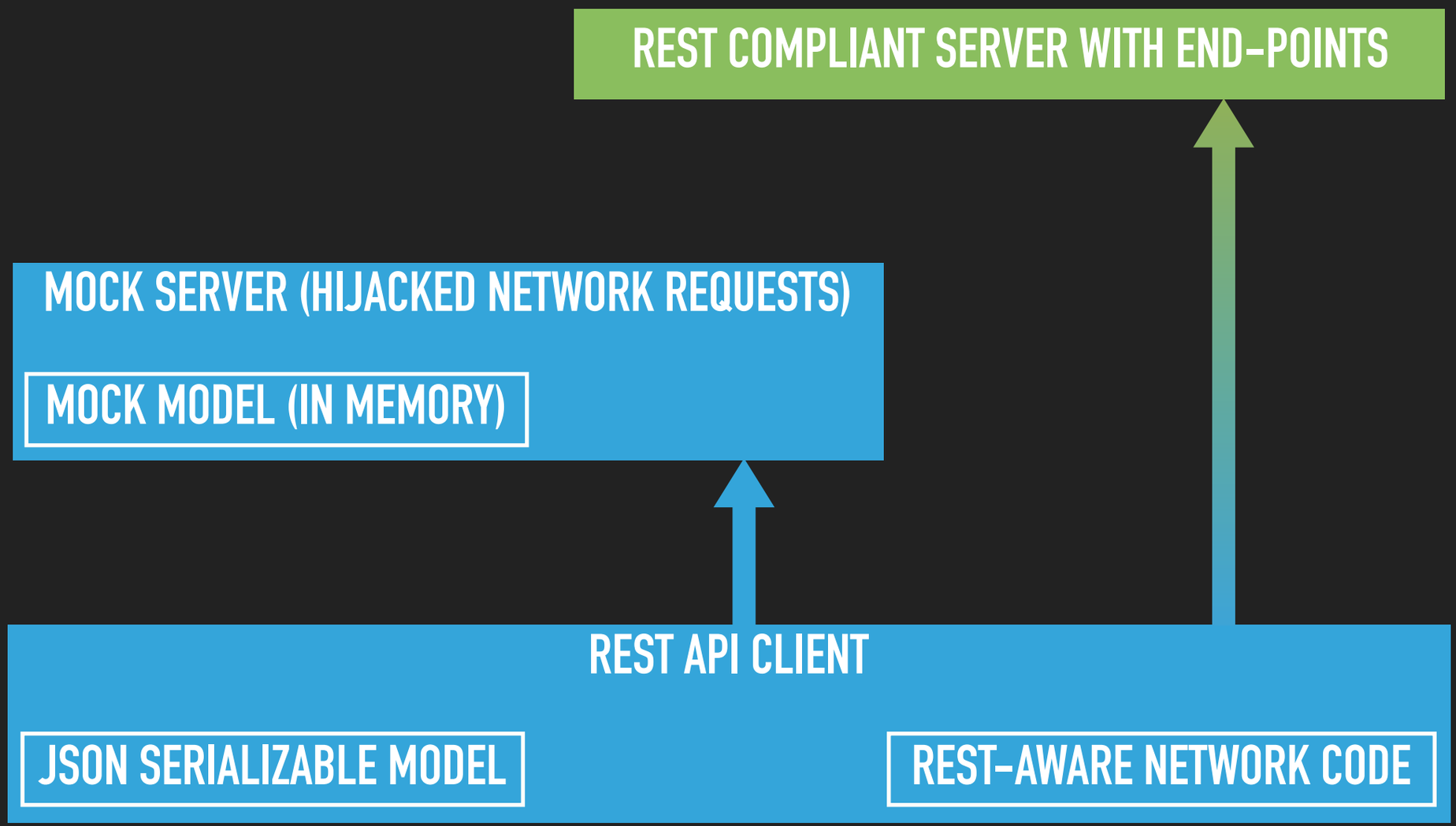


# TESTING THE MODEL LAYER IS DELICIOUSLY EASY

- Test serialization / deserialization
  - Handling of Optional properties
- Test Equality and Comparison (==, <, etc.)
- Test ‘convenience’ properties

# NETWORK SERVICE API LAYER

- Asynchronous REST service calls in background



# MOCK THREE ENDPOINTS

## USER

/api/user
  one: /<nsuuid>

GET (retrieve)
  all returns [User]
  one returns User

POST (create)
  body contains User
  returns User with ID

PUT (update)
  body contains User
  returns User

DELETE (remove)
  must specify one

## GAME

/api/game
  one: /<nsuuid>

GET (retrieve)
  all returns [Game]
  one returns Game

POST (create)
  body contains Game
  returns Game with ID

PUT (update)
  body contains Game
  returns Game

DELETE (remove)
  must specify one

## MESSAGE

/api/message
  one: /<nsuuid>

GET (retrieve)
  all returns [Message]
  one returns Message

POST (create)
  body contains Message
  returns Message with ID

PUT (update)
  body contains Message
  returns Message

DELETE (remove)
  must specify one

# GENERIC ENDPOINT MOCK

**T : MODELITEM**

/api/<model-item-name>
  one: /<nsuuid>

GET (retrieve)
  all returns [T]
  one returns T

POST (create)
  body contains T
  returns T with ID set

PUT (update)
  body contains T
  returns T

DELETE (remove)
  must specify one

Uses OHHTTP stubs to intercept HTTP request

Manages items store [T] based on request

Can be assigned the logged in user to filter or authorize activity

RSDRESTServices CocoaPod

YOU HAVE ENGAGED MY VALUABLE SERVICES, YOUR MAJESTY. JUST TELL ME WHERE I CAN FIND THIS OGRE.

---

# MOCK SERVER BENEFITS

- Can now do the following without spinning up a server
  - Develop the app and run against test data
  - Targeted QA testing (against crafted data sets)
  - Run UI tests on a test server

# TESTING ASYNCHRONOUS API CALLS

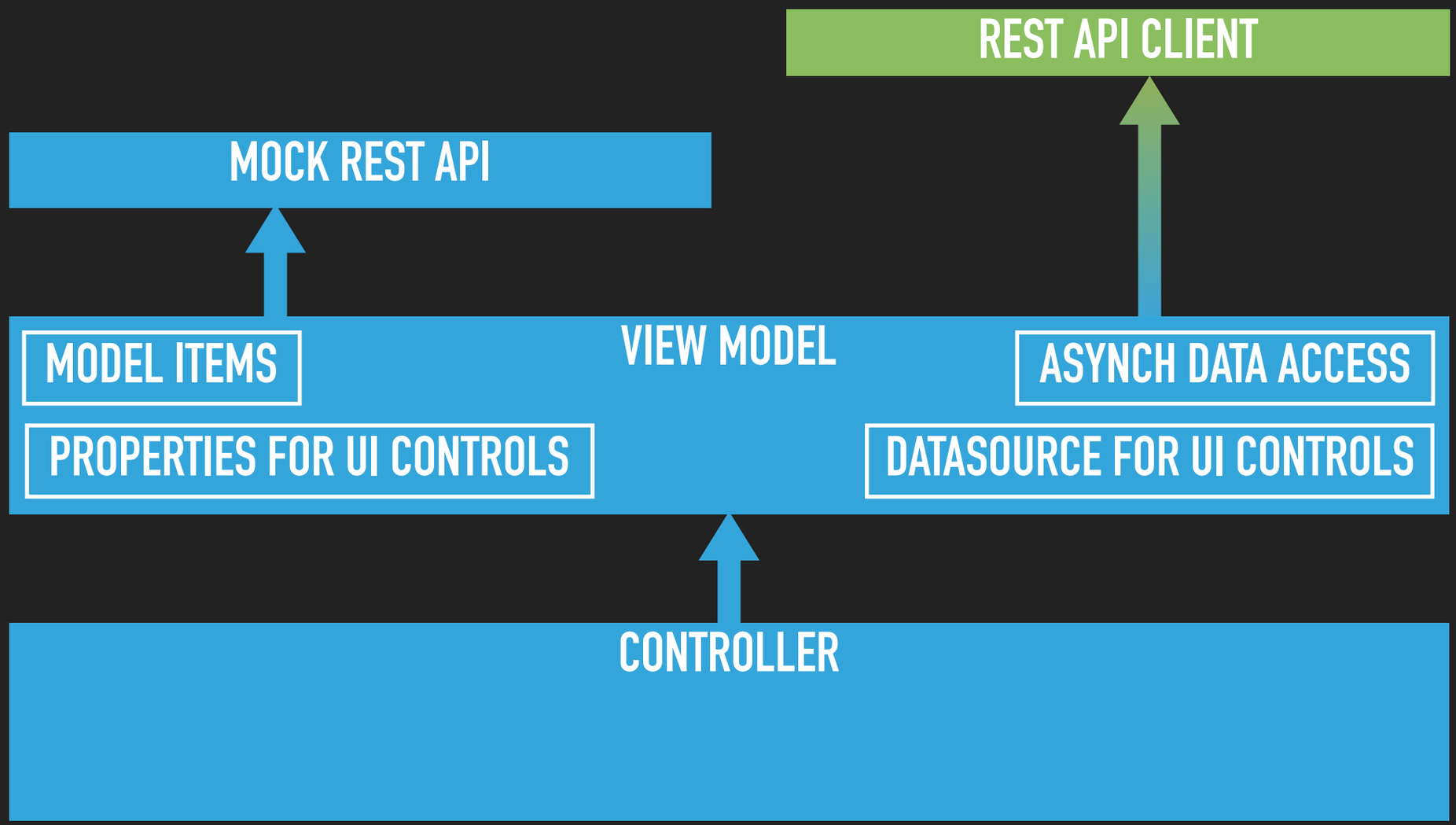
- Execute async Api call with callback closure
  - Capture parameters passed to callback closure
  - Set self.called = true at end of closure
- After this statement, the callback has not completed
  - Need to wait until self.called is set

# WAIT FOR ASYNCHRONOUS ACTIVITY TO COMPLETE

- waitForResponse { self.called }
  - Executes runLoop & periodically executes callback
  - Return success if callback true, fail after 5 seconds
- After waitForResponse, test is ‘synchronous’ again
  - Can test values of captured variables
  - Safe to test state of other objects as well

# VIEW MODEL LAYER

- Reformats model data needed by the controller
- Responds to Controller



FIONA: WHAT KIND OF KNIGHT ARE YOU?
SHREK: ONE OF A KIND.

# VIEW MODEL DESIGN AND RESPONSIBILITIES

- VM has a Protocol so it too can be mocked
- Each VM owned by only one Controller
- UI control properties assigned from the view model
- VM acts as DataSource for UI Tables, Collections, etc.
- VM makes async REST service calls
  - Pushes results to UI Thread

# ASYNC API CALLS

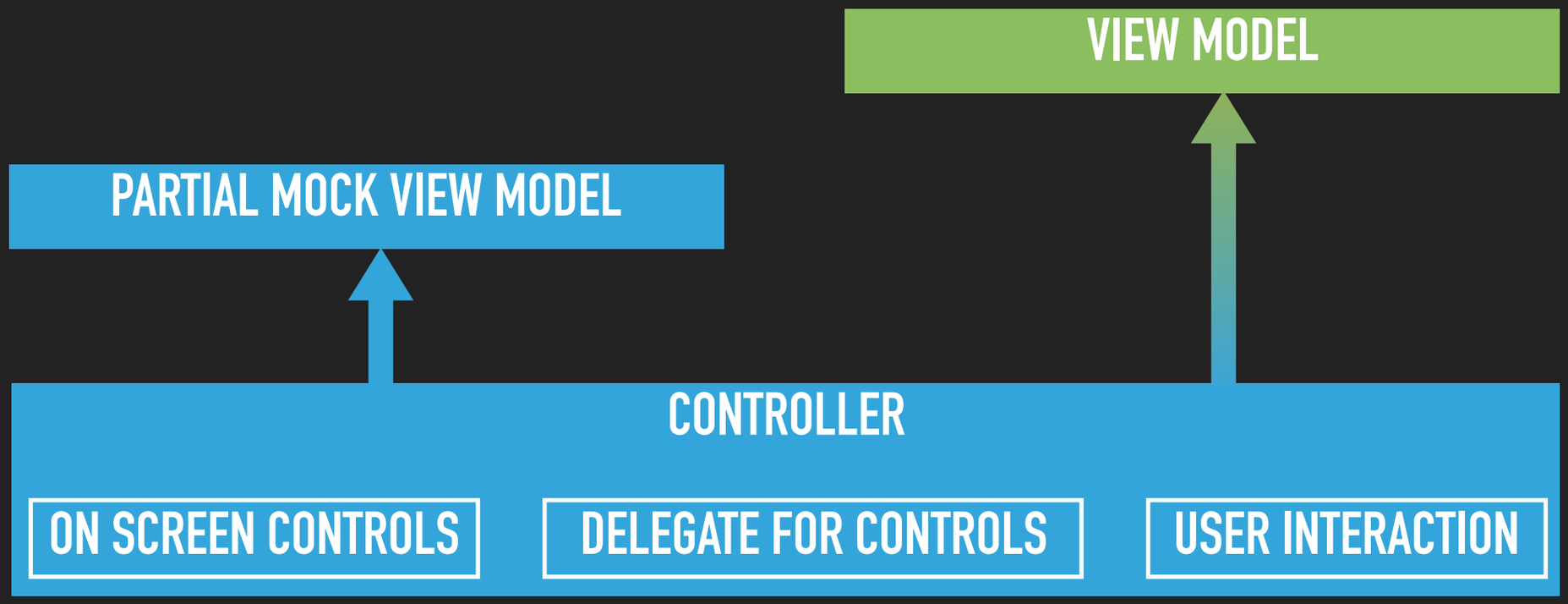
- Methods that make REST API calls
  - Call given closure that's called on a background thread
  - Update VM state on the background thread
  - Respond to caller with completionHandler on UI thread
- Uses fireOnMainThread function wrapper

# TESTING VIEW MODELS

- Initialization should mock Api singleton
  - Supplies fake responses on background thread
- Execute VM method that uses Api - capture callback results
- waitForResponse
  - Check error result
  - Check VM updated internal state
  - Check VM pushed response to UI thread

# CONTROLLER LAYER

- Display UI Controls with data
- Accept user input



# TIMING ISSUES IN VIEW CONTROLLERS

- UI Elements cannot be populated until BOTH:
  - viewDidLoad completes
  - loadData completes
- initializeComponentsFromViewModel
  - Called at end of both viewDidLoad and loadData
  - Guard clauses for View loaded and VM Loaded

# TESTING CONTROLLERS

- In test setUp
  - resurrect controller from storyboard
  - set controller.view.hidden = false (force viewDidLoad)
    - Replace controller's viewModel with Partial Mock
      - If this view model has async behaviors it implements
- In test (or test setUp)
  - call loadData to load fake data

# PARTIAL MOCK OF VIEW MODEL

- Example: GamesControllerTests - testDeleteGame
- Partial mock implements the view model protocol
- Is passed the actual view model in its constructor
- Delegates properties and non-async functions to contained VM
- Async functions are mocked to return fake data synchronously

# HANDLING TRANSITIONS IN TESTS

- Goal - Support Apple-standard ways of designing UI
  - Use Interface Builder to design screens
  - Use Segues to transition between screens
- Desired features
  - Intercept segue attempt and prevent segue
  - Intercept segue attempt but allow segue to continue
  - Use closures to keep intercept code local to test

# SWIZZLING: FREAKY FRIDAY MADE FREAKIER

- Swaps bodies of two methods with the same signature
- Use extensions to add the test method to your Controller
- Use Swift Generics to write this code only once
- RSDTesting CocoaPod
  - prepareForSegue, presentViewController, etc.
- But what I REALLY want is to swizzle in a closure

# DANCE STEPS TO SWIZZLE IN AN INTERCEPTOR METHOD

- In test fixture setup: swizzle the real method with the test method
- In your test: Create a boxed callback intercept function
  - Accepts the parameters of method you are swizzling
  - Returns Bool to indicate if the real method should also run
  - Add boxed callback to your Controller using extensions
- In the swizzled method:  Retrieve the boxed callback function
  - Execute callback, capture result (true if callback not set)
  - Calls real method based on the result

# SEGUES I WOULD LIKE TO TEST

- Interact with AlertController to create a new game
- ShowGames segues to MessageList for selected game
- UpdateUsers pops Save dialog on back-button press
- ShowUsers segues to UpdateUsers for selected user
  - test delete user from update screen deletes the row

# MOCK CLICKING ACTIONS ON AN ACTION CONTROLLER

- Test AlertController displays when exiting user update
  - Click “save”, “cancel”, or “exit without saving”
- You can pull the list of actions from the controller
  - But you can’t execute the action
  - Even through it was just a closure
- Here’s one (hack) that doesn’t cruft runtime too badly

# THE TROUBLE WITH TESTING UI

- Asynchronous behavior in the UI
  - Mocking the VM makes data retrieval synchronous
  - But there is still Core Animation
  - And UIView Animation

# EMBRACE THE ASYNCHRONY

- Handle Core Animation callbacks
  - CATransaction and runLoop hook
    - waitForTransactions in tearDown
  - Prevents CA from calling back into a dead test
- Also Handled in AsynchronousTestCase
  - In RSDTesting CocoaPod

# THE TROUBLE WITH TESTING UI

- Private UI controls need to be accessible by the test
  - To examine labels
  - To type in text fields
  - To scroll, swipe, and click on various UI controls
  - To navigate between screens

# APPLE HAS PROVIDED THE @TESTABLE IMPORT

- UI Controls no longer need to be public to be tested
  - This is NOT just for UI testing
- Things you are testing still need to be internal
  - Private things are still invisible to your test suite

# THE TROUBLE WITH TESTING UI

- Some hardware-style events are hard to simulate
  - Rotate
  - Shake
  - Push notifications

# UI TESTING TO THE RESCUE?

- UI testing is too slow and buggy for ALL testing
- But there are some things that it can be used for:
  - Simulating hardware events
    - Rotation, shake, etc.
  - Moving from screen to screen and interacting
- Enable mock HTTP server on UI test execution

# WWDC (2015) VIDEOS AND OTHER GOODNESS

- https://developer.apple.com/videos/play/wwdc2015-414/
- https://developer.apple.com/videos/play/wwdc2015-406/
- https://developer.apple.com/videos/play/wwdc2015-104/
- http://nshipster.com/swift-objc-runtime/
- https://www.objc.io/issues/13-architecture/mvvm/

I JUST KNOW, BEFORE THIS IS OVER, I'M GONNA NEED A WHOLE LOT OF SERIOUS THERAPY. LOOK AT MY EYE TWITCHIN'.

---

# THANK YOU

- Ravi Desai
- RaviDesai@github.com
  - /CodeMash2016
    - pod repo add RSDSpecs http://github.com/RaviDesai/RSDSpecs.git
    - /RSDTesting, /RSDSerialization, /RSDRESTServices
- ravi.s.desai@gmail.com
- @ravi_s_desai