Mastering MVVM With Swift

12 Supercharging MVVM With Protocols

Let’s apply what we learned in the previous chapters to the week view

controller. The week view controller currently configures the cells of its

table view. That’s something we want to change. The refactoring of the

week view controller involves four steps:

We create a view model for each table view cell.

The WeekViewViewModel generates a view model for each table view

cell.

We define a protocol which the view models of the table view cells

conform to.

The WeatherDayTableViewCell is be able to configure itself using a

view model.

Creating a New View Model

We first create a view model. Create a file in the View Models group of

the Weather View Controllers group and name it

WeatherDayViewViewModel.swift.

Creating a New View Model

Replace the import statement for Foundation with an import statement

for UIKit and define the WeatherDayViewViewModel struct.

WeatherDayViewViewModel.swift

1 import UIKit

2

3 struct WeatherDayViewViewModel {

4

5 }

The model the view model will manage is an instance of the

WeatherDayData struct.

WeatherDayViewViewModel.swift

1 import UIKit

2

3 struct WeatherDayViewViewModel {

4

5 // MARK: - Properties

6

7 let weatherDayData: WeatherDayData

8

9 }

We can migrate most of the code from the WeekViewViewModel struct to

the WeatherDayViewViewModel struct. The most important differences are

that we use computed properties and no longer need to fetch a model

from an array of models. This is what the WeatherDayViewViewModel struct

looks like.

WeatherDayViewViewModel.swift

1 import UIKit

2

3 struct WeatherDayViewViewModel {

4

5 // MARK: - Properties

6

7 let weatherDayData: WeatherDayData

8

9 // MARK: -

10

11 private let dayFormatter = DateFormatter()

12 private let dateFormatter = DateFormatter()

13

14 // MARK: - Public Interface

15

16 var day: String {

17 // Configure Date Formatter

18 dayFormatter.dateFormat = "EEEE"

19

20 return dayFormatter.string(from: weatherDayData.time)

21 }

22

23 var date: String {

24 // Configure Date Formatter

25 dateFormatter.dateFormat = "MMMM d"

26

27 return dateFormatter.string(from: weatherDayData.time)

28 }

29

30 var temperature: String {

31 let min = format(temperature: weatherDayData.temperatureMin)

32 let max = format(temperature: weatherDayData.temperatureMax)

33

34 return "\(min) - \(max)"

35 }

36

37 var windSpeed: String {

38 let windSpeed = weatherDayData.windSpeed

39

40 switch UserDefaults.unitsNotation() {

41 case .imperial:

42 return String(format: "%.f MPH", windSpeed)

43 case .metric:

44 return String(format: "%.f KPH", windSpeed.toKPH())

45 }

46 }

47

48 var image: UIImage? {

49 return UIImage.imageForIcon(withName: weatherDayData.icon)

50 }

51

52 // MARK: - Private Interface

53

54 private func format(temperature: Double) -> String {

55 switch UserDefaults.temperatureNotation() {

56 case .fahrenheit:

57 return String(format: "%.0f F", temperature)

58 case .celsius:

59 return String(format: "%.0f C", temperature.toCelcius())

60 }

61 }

62

63 }

This should look familiar. We now have a view model that we can use to

populate a WeatherDayTableViewCell instance.

Refactoring the Week View View Model

The next step involves drastically refactoring the WeekViewViewModel.

Everything needs to go with the exception of the weatherData,

numberOfSections, and numberOfDays properties.

WeekViewViewModel.swift

1 import UIKit

2

3 struct WeekViewViewModel {

4

5 // MARK: - Properties

6

7 let weatherData: [WeatherDayData]

8

9 // MARK: -

10

11 var numberOfSections: Int {

12 return 1

13 }

14

15 var numberOfDays: Int {

16 return weatherData.count

17 }

18

19 }

Because the week view view model is now responsible for supplying a

view model for each table view cell of the week view controller, we need

to implement a new method, viewModel(for:). This method takes an

index as its only argument. The index corresponds with a row in the table

view of the week view controller. The viewModel(for:) method returns an

instance of the WeatherDayViewViewModel struct.

WeekViewViewModel.swift

1 func viewModel(for index: Int) -> WeatherDayViewViewModel {

2 return WeatherDayViewViewModel(weatherDayData: weatherData[index\

3 ])

4 }

In the viewModel(for:) method, we fetch the WeatherDayData instance that

corresponds with the value of the index argument and use it to create an

instance of the WeatherDayViewViewModel struct.

Creating Another Protocol

We could pass the WeatherDayViewViewModel instance directly to the table

view cell, but, as I explained earlier, I prefer to use a protocol to define

the interface the table view cell expects. Remember that this adds a layer

of abstraction between the view model layer and the view layer.

Create a new group in the Weather View Controllers group and name it

Protocols.

Creating the Protocols Group

Create a file for the protocol and name it

WeatherDayRepresentable.swift.

Creating WeatherDayRepresentable.swift

Replace the import statement for Foundation with an import statement

for UIKit and declare the WeatherDayRepresentable protocol.

WeatherDayRepresentable.swift

1 import UIKit

2

3 protocol WeatherDayRepresentable {

4

5 }

The WeatherDayRepresentable protocol declares five properties:

day of type String

date of type String,

image of type UIImage?

windSpeed of type String

temperature of type String

WeatherDayRepresentable.swift

1 import UIKit

2

3 protocol WeatherDayRepresentable {

4

5 var day: String { get }

6 var date: String { get }

7 var image: UIImage? { get }

8 var windSpeed: String { get }

9 var temperature: String { get }

10

11 }

We need to make sure that the WeatherDayTableViewCell class knows

how to handle an instance of the WeatherDayViewViewModel struct. To

accomplish that, the view model needs to adopt the

WeatherDayRepresentable protocol. This is very easy.

Open WeatherDayViewViewModel.swift and create an extension for

the WeatherDayViewViewModel struct. We use the extensions to conform

the WeatherDayViewViewModel struct to the WeatherDayRepresentable

protocol.

WeatherDayViewViewModel.swift

1 import UIKit

2

3 struct WeatherDayViewViewModel {

4

5 ...

6

7 }

8

9 extension WeatherDayViewViewModel: WeatherDayRepresentable {

10

11 }

Because the WeatherDayViewViewModel struct implicitly conforms to the

WeatherDayRepresentable protocol, that’s all we need to do.

Updating the Weather Day Table View Cell

Last but not least, we need to implement a new method in the

WeatherDayTableViewCell class. Open WeatherDayTableViewCell.swift

and define a new method named configure(withViewModel:). In the

body, we configure the subviews of the table view cell, using the view

model. This should look very familiar by now.

WeatherDayTableViewCell.swift

1 func configure(withViewModel viewModel: WeatherDayRepresentable) {

2 dayLabel.text = viewModel.day

3 dateLabel.text = viewModel.date

4 iconImageView.image = viewModel.image

5 windSpeedLabel.text = viewModel.windSpeed

6 temperatureLabel.text = viewModel.temperature

7 }

Updating the Week View Controller

All that remains to be done is updating the tableView(\_:cellForRowAt:)

method of the week view controller. We ask the view model of the week

view controller to create a view model for a particular index. If we receive

a view model, we pass it to the configure(withViewModel:) method of the

WeatherDayTableViewCell.

WeekViewController.swift

1 func tableView(\_ tableView: UITableView, cellForRowAt indexPath: Ind\

2 exPath) -> UITableViewCell {

3 guard let cell = tableView.dequeueReusableCell(withIdentifier: W\

4 eatherDayTableViewCell.reuseIdentifier, for: indexPath) as? WeatherD\

5 ayTableViewCell else { fatalError("Unexpected Table View Cell") }

6

7 if let weatherDayRepresentable = viewModel?.viewModel(for: index\

8 Path.row) {

9 cell.configure(withViewModel: weatherDayRepresentable)

10 }

11

12 return cell

13 }

The implementation of the UITableViewDataSource protocol has

undergone a dramatic transformation. The week view controller has no

clue about the data the Dark Sky API returns to the application. It uses a

view model to populate its table view and that’s all it does, apart from

responding to events, such as updating the user interface when it

receives a new view model.

I hope that the past few chapters have convinced you of the benefits of

the Model-View-ViewModel pattern. Not only does it result in a clear

separation of responsibilities, the testability of the project has improved

substantially. And that’s something we look at in the next chapters.

13 Ready, Set, Test

Because we moved a fair bit of logic from the view controllers of the

project into the view models, we gained an important advantage,

improved testability. As I mentioned earlier, unit testing view controllers

is known to be difficult. View models, however, are easy to test. And

that’s what I’ll show you in the next few chapters.

Adding a Unit Test Target

Before we can start testing the view models, we need to add a target for

the unit tests. Select the project in the Project Navigator, click the plus

button at the bottom, and choose iOS Unit Testing Bundle.

Adding a Target

Choosing the iOS Unit Testing Bundle Template

The defaults are just fine. Make sure Language is set to Swift and

Target to be Tested is set to Cloudy.

Configuring the Target

The Cloudy project should now have two targets, Cloudy and

CloudyTests.

The Cloudy project should now have two targets.

Organizing the Unit Test Target

I usually create several groups in the unit testing bundle to keep files and

folders organized. I create a group named Supporting Files for the

Info.plist file, a group for stubs, a group for extensions, and a group for

the test cases. This is what you should end up with. I also removed the

test case Xcode created for us, CloudyTests.swift. We’re going to start

from scratch.

Organizing the Unit Test Target

If you take this approach and move the Info.plist file in the Supporting

Files group, make sure you update the file reference in Xcode. The

Info.plist file shouldn’t appear red in the Project Navigator.

We also need to update the path to the Info.plist file in the build settings

of the CloudyTests target. Choose the CloudyTests target from the list

of targets, select Build Settings at the top, and search for the Info.plist

File build setting in the Packaging section. Change the path from

CloudyTests/Info.plist to CloudyTests/Supporting Files/Info.plist.

Updating the Build Settings

To make sure the unit test target is correctly configured, we need to run

the test suite. We don’t have any unit tests yet, but that’s not a problem.

We only verify that the unit test target is ready and properly configured.

Choose a simulator from the list of devices and run the test suite by

choosing Test from Xcode’s Product menu. The application is installed

in the simulator and the test suite is run. No errors or warnings should be

visible.

Running the Test Suite

In the next chapter, we write unit tests for the view models of the settings

view controller.

14 Testing Your First View Model

Creating a Test Case

In this chapter, we test the view models of the settings view controller.

We start with the SettingsViewTimeViewModel struct. Create a new file in

the Test Cases group we created in the previous chapter and choose the

Unit Test Case Class template.

Choosing the Unit Test Case Class Template

You can name the file whatever you want, but I usually use the name of

the type I am testing followed by the suffix Tests. This means I name the

file SettingsViewTimeViewModelTests.swift. It’s a bit long, but it’s very

descriptive.

Creating SettingsViewTimeViewModelTests.swift

Click Create to create the file. It’s possible that Xcode shows you a

dialog when you create your first test case. Xcode offers you to create an

Objective-C bridging header. This isn’t necessary. Click Don’t Create to

dismiss the dialog.

There’s no need for an Objective-C bridging header.

Importing the Cloudy Module

To access the code from the Cloudy target, we need to add an import

statement for the Cloudy module. To make sure we can access internal

entities, we prefix the import statement with the testable attribute.

SettingsViewTimeViewModelTests.swift

1 import XCTest

2 @testable import Cloudy

3

4 class SettingsViewTimeViewModelTests: XCTestCase {

5

6 ...

7

8 }

Remove any existing unit tests and remove the comments from the

setUp() and tearDown() methods. I’d like to start with a clean slate.

SettingsViewTimeViewModelTests.swift

1 import XCTest

2 @testable import Cloudy

3

4 class SettingsViewTimeViewModelTests: XCTestCase {

5

6 // MARK: - Set Up & Tear Down

7

8 override func setUp() {

9 super.setUp()

10 }

11

12 override func tearDown() {

13 super.tearDown()

14 }

15

16 }

Writing a Unit Test

The first unit test we’re going to write tests the text computed property of

the SettingsViewTimeViewModel. Revisit the implementation of the

SettingsViewTimeViewModel struct if you need to freshen up your memory.

Because the text computed property can return two possible values, we

need to write two unit tests for complete test coverage.

SettingsViewTimeViewModel.swift

1 var text: String {

2 switch timeNotation {

3 case .twelveHour: return "12 Hour"

4 case .twentyFourHour: return "24 Hour"

5 }

6 }

The first test is very simple. We instantiate a SettingsViewTimeViewModel

instance and pass in the model as an argument, a TimeNotation instance.

To test the text computed property, we need to assert that the value text

returns is equal to 12 Hour. That’s it. Very simple.

SettingsViewTimeViewModelTests.swift

1 // MARK: - Tests for Text

2

3 func testText\_TwelveHour() {

4 let viewModel = SettingsViewTimeViewModel(timeNotation: .twelveH\

5 our)

6

7 XCTAssertEqual(viewModel.text, "12 Hour")

8 }

We have several options to run the unit test we just created. To run every

unit test of the test suite, choose Test from Xcode’s Product menu or

press Command + U. We can also click the diamonds in the gutter of the

code editor. If we click the diamond next to the class definition, every unit

test of the XCTestCase subclass is run.

Running Unit Tests of a XCTestCase Subclass

To run one unit test, we click the diamond next to the test we’re

interested in.

Running Individual Unit Tests

Press Command + U to run the test suite. The diamonds in the gutter of

the editor should turn green, indicating that the unit test has passed.

Make sure that Destination is set to one of the simulators because

Xcode currently doesn’t support running a test suite with a physical

device as the destination.

The unit test passed.

The unit test passed.

The second test for the text computed property is almost identical. The

only changes we need to make are the model we pass to the initializer of

the view model and the assertion of the unit test. The result returned by

the text computed property should be equal to 24 Hour.

SettingsViewTimeViewModelTests.swift

1 func testText\_TwentyFourHour() {

2 let viewModel = SettingsViewTimeViewModel(timeNotation: .twentyF\

3 ourHour)

4

5 XCTAssertEqual(viewModel.text, "24 Hour")

6 }

Press Command + U one more time to run the test suite to make sure

the unit tests pass. That’s looking good.

The unit tests passed.

You’ve probably noticed that I use a specific convention for naming the

test methods. Each method starts with the word test, which is required,

followed by the name of the method or computed property.

Whenever I write multiple unit tests for a single method or computed

property, I append a descriptive keyword to the method’s name, using an

underscore for readability. This is a personal choice that I like because it

makes the test methods easier to read. If you name your unit tests

methods, testText1() and testText2(), you need to read the

implementation of the unit test to understand how they differ. Give it a try

and see if you like it.

SettingsViewTimeViewModelTests.swift

1 import XCTest

2 @testable import Cloudy

3

4 class SettingsViewTimeViewModelTests: XCTestCase {

5

6 // MARK: - Set Up & Tear Down

7

8 override func setUp() {

9 super.setUp()

10 }

11

12 override func tearDown() {

13 super.tearDown()

14 }

15

16 // MARK: - Tests for Text

17

18 func testText\_TwelveHour() {

19 ...

20 }

21

22 func testText\_TwentyFourHour() {

23 ...

24 }

25

26 }

Writing More Unit Tests

The unit tests for the accessoryType computed property are a bit more

complex because the user defaults database is accessed in the body of

the computed property. Take a look at the implementation of the

accessoryType computed property of the SettingsViewTimeViewModel

struct.

SettingsViewTimeViewModel.swift

1 var accessoryType: UITableViewCellAccessoryType {

2 if UserDefaults.timeNotation() == timeNotation {

3 return .checkmark

4 } else {

5 return .none

6 }

7 }

The user’s setting, which is stored in the user defaults database, can

have one of two values. The model of the view model can also have one

have one of two values. The model of the view model can also have one

of two values. This means we have to write four units tests for complete

test coverage.

The name of the first test method,

testAccessoryType\_TwelveHour\_TwelveHour(), shows what I mean. The

first suffix, TwelveHour, hints at the value stored in the user defaults

database. The second suffix, TwelveHour, hints at the value of the model

of the view model.

SettingsViewTimeViewModelTests.swift

1 // MARK: - Tests for Accessory Type

2

3 func testAccessoryType\_TwelveHour\_TwelveHour() {

4

5 }

Despite this complexity, the unit test itself is fairly simple. We create a

TimeNotation instance and use it to update the user defaults database.

We then create a SettingsViewTimeViewModel instance by passing in

another TimeNotation instance.

SettingsViewTimeViewModelTests.swift

1 // MARK: - Tests for Accessory Type

2

3 func testAccessoryType\_TwelveHour\_TwelveHour() {

4 let timeNotation: TimeNotation = .twelveHour

5 UserDefaults.standard.set(timeNotation.rawValue, forKey: UserDef\

6 aultsKeys.timeNotation)

7

8 let viewModel = SettingsViewTimeViewModel(timeNotation: .twelveH\

9 our)

10 }

If the user’s preference in the user defaults database is twelve hour time

notation and the model of the view model is also twelve hour time

notation, then the accessory type returned by the accessoryType

computed property should be equal to the

UITableViewCellAccessoryType.checkmark member. This is reflected in the

assertion of the unit test.

SettingsViewTimeViewModelTests.swift

1 // MARK: - Tests for Accessory Type

2

3 func testAccessoryType\_TwelveHour\_TwelveHour() {

4 let timeNotation: TimeNotation = .twelveHour

5 UserDefaults.standard.set(timeNotation.rawValue, forKey: UserDef\

6 aultsKeys.timeNotation)

7

8 let viewModel = SettingsViewTimeViewModel(timeNotation: .twelveH\

9 our)

10

11 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAccessory\

12 Type.checkmark)

13 }

Press Command + U to run the unit tests of the

SettingsViewTimeViewModel struct to make sure they all pass. Open the

Test Navigator on the right to bring up an overview of the unit test

results.

Bringing Up the Test Navigator

The other three unit tests are permutations of this scenario. Put the book

aside for a moment and try implementing the other three unit tests

yourself. That’s the best way to learn how to write unit tests. This is what

the unit tests should look like when you’re finished.

SettingsViewTimeViewModelTests.swift

1 import XCTest

2 @testable import Cloudy

3

4 class SettingsViewTimeViewModelTests: XCTestCase {

5

6 ...

7

8 // MARK: - Tests for Accessory Type

9

10 func testAccessoryType\_TwelveHour\_TwelveHour() {

11 let timeNotation: TimeNotation = .twelveHour

12 UserDefaults.standard.set(timeNotation.rawValue, forKey: Use\

13 rDefaultsKeys.timeNotation)

14

15 let viewModel = SettingsViewTimeViewModel(timeNotation: .twe\

16 lveHour)

17

18 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\

19 soryType.checkmark)

20 }

21

22 func testAccessoryType\_TwelveHour\_TwentyFourHour() {

23 let timeNotation: TimeNotation = .twelveHour

24 UserDefaults.standard.set(timeNotation.rawValue, forKey: Use\

25 rDefaultsKeys.timeNotation)

26

27 let viewModel = SettingsViewTimeViewModel(timeNotation: .twe\

28 ntyFourHour)

29

30 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\

31 soryType.none)

32 }

33

34 func testAccessoryType\_TwentyFourHour\_TwelveHour() {

35 let timeNotation: TimeNotation = .twentyFourHour

36 UserDefaults.standard.set(timeNotation.rawValue, forKey: Use\

37 rDefaultsKeys.timeNotation)

38

39 let viewModel = SettingsViewTimeViewModel(timeNotation: .twe\

40 lveHour)

41

42 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\

43 soryType.none)

44 }

45

46 func testAccessoryType\_TwentyFourHour\_TwentyFourHour() {

47 let timeNotation: TimeNotation = .twentyFourHour

48 UserDefaults.standard.set(timeNotation.rawValue, forKey: Use\

49 rDefaultsKeys.timeNotation)

50

51 let viewModel = SettingsViewTimeViewModel(timeNotation: .twe\

52 ntyFourHour)

53

54 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\

55 soryType.checkmark)

56 }

57

58 }

Running the Test Suite

Resetting State

Because we modify a value of the user defaults database in the unit

tests, it’s important that we reset that state after each unit test. We can

do this in the tearDown() method of the XCTestCase subclass.

We remove the object for the key used to store the value of the time

notation. Even though it isn’t strictly necessary for these unit tests, it’s a

good practice to always reset the state you set or modify in a unit test.

SettingsViewTimeViewModelTests.swift

1 override func tearDown() {

2 super.tearDown()

3

4 // Reset User Defaults

5 UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.time\

6 Notation)

7 }

If you’re not familiar with unit testing, the setUp() method is invoked

before a unit test is run and the tearDown() method is invoked after a unit

test is run. In other words, the setUp() and tearDown() methods are

invoked six times because we’ve written six unit tests.

Remember that, if the logic contained in the SettingsViewTimeViewModel

was still in the SettingsViewController class, the unit tests would be

much more complex. I hope you can see that unit testing a view model

isn’t difficult. It’s much easier than unit testing a view controller.

Unit Testing the Other View Models

The unit tests for the SettingsViewUnitsViewModel struct and the

SettingsViewTemperatureViewModel struct are very similar to those of the

SettingsViewTimeViewModel struct. Put the book aside for a moment and

try to implement the unit tests for the other two view models. You can find

the solution below.

SettingsViewUnitsViewModelTests.swift

1 import XCTest

2 @testable import Cloudy

3

4 class SettingsViewUnitsViewModelTests: XCTestCase {

5

6 // MARK: - Set Up & Tear Down

7

8 override func setUp() {

9 super.setUp()

10 }

11

12 override func tearDown() {

13 super.tearDown()

14

15 // Reset User Defaults

16 UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.\

17 unitsNotation)

18 }

19

20 // MARK: - Tests for Text

21

22 func testText\_Imperial() {

23 let viewModel = SettingsViewUnitsViewModel(unitsNotation: .i\

24 mperial)

25

26 XCTAssertEqual(viewModel.text, "Imperial")

27 }

28

29 func testText\_Metric() {

30 let viewModel = SettingsViewUnitsViewModel(unitsNotation: .m\

31 etric)

32

33 XCTAssertEqual(viewModel.text, "Metric")

34 }

35

36 // MARK: - Tests for Accessory Type

37

38 func testAccessoryType\_Imperial\_Imperial() {

39 let unitsNotation: UnitsNotation = .imperial

40 UserDefaults.standard.set(unitsNotation.rawValue, forKey: Us\

41 erDefaultsKeys.unitsNotation)

42

43 let viewModel = SettingsViewUnitsViewModel(unitsNotation: .i\

44 mperial)

45

46 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\

47 soryType.checkmark)

48 }

49

50

51 func testAccessoryType\_Imperial\_Metric() {

52 let unitsNotation: UnitsNotation = .imperial

53 UserDefaults.standard.set(unitsNotation.rawValue, forKey: Us\

54 erDefaultsKeys.unitsNotation)

55

56 let viewModel = SettingsViewUnitsViewModel(unitsNotation: .m\

57 etric)

58

59 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\

60 soryType.none)

61 }

62

63 func testAccessoryType\_Metric\_Imperial() {

64 let unitsNotation: UnitsNotation = .metric

65 UserDefaults.standard.set(unitsNotation.rawValue, forKey: Us\

66 erDefaultsKeys.unitsNotation)

67

68 let viewModel = SettingsViewUnitsViewModel(unitsNotation: .i\

69 mperial)

70

71 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\

72 soryType.none)

73 }

74

75

76 func testAccessoryType\_Metric\_Metric() {

77 let unitsNotation: UnitsNotation = .metric

78 UserDefaults.standard.set(unitsNotation.rawValue, forKey: Us\

79 erDefaultsKeys.unitsNotation)

80

81 let viewModel = SettingsViewUnitsViewModel(unitsNotation: .m\

82 etric)

83

84 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\

85 soryType.checkmark)

86 }

87

88 }

SettingsViewTemperatureViewModelTests.swift

1 import XCTest

2 @testable import Cloudy

3

4 class SettingsViewTemperatureViewModelTests: XCTestCase {

5

6 // MARK: - Set Up & Tear Down

7

8 override func setUp() {

9 super.setUp()

10 }

11

12 override func tearDown() {

13 super.tearDown()

14

15 // Reset User Defaults

16 UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.\

17 temperatureNotation)

18 }

19

20 // MARK: - Tests for Text

21

22 func testText\_Fahrenheit() {

23 let viewModel = SettingsViewTemperatureViewModel(temperature\

24 Notation: .fahrenheit)

25

26 XCTAssertEqual(viewModel.text, "Fahrenheit")

27 }

28

29 func testText\_Celsius() {

30 let viewModel = SettingsViewTemperatureViewModel(temperature\

31 Notation: .celsius)

32

33 XCTAssertEqual(viewModel.text, "Celsius")

34 }

35

36 // MARK: - Tests for Accessory Type

37

38 func testAccessoryType\_Fahrenheit\_Fahrenheit() {

39 let temperatureNotation: TemperatureNotation = .fahrenheit

40 UserDefaults.standard.set(temperatureNotation.rawValue, forK\

41 ey: UserDefaultsKeys.temperatureNotation)

42

43 let viewModel = SettingsViewTemperatureViewModel(temperature\

44 Notation: .fahrenheit)

45

46 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\

47 soryType.checkmark)

48 }

49

50 func testAccessoryType\_Fahrenheit\_Celsius() {

51 let temperatureNotation: TemperatureNotation = .fahrenheit

52 UserDefaults.standard.set(temperatureNotation.rawValue, forK\

53 ey: UserDefaultsKeys.temperatureNotation)

54

55 let viewModel = SettingsViewTemperatureViewModel(temperature\

56 Notation: .celsius)

57

58 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\

59 soryType.none)

60 }

61

62 func testAccessoryType\_Celsius\_Fahrenheit() {

63 let temperatureNotation: TemperatureNotation = .celsius

64 UserDefaults.standard.set(temperatureNotation.rawValue, forK\

65 ey: UserDefaultsKeys.temperatureNotation)

66

67 let viewModel = SettingsViewTemperatureViewModel(temperature\

68 Notation: .fahrenheit)

69

70 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\

71 soryType.none)

72 }

73

74 func testAccessoryType\_Celsius\_Celsius() {

75 let temperatureNotation: TemperatureNotation = .celsius

76 UserDefaults.standard.set(temperatureNotation.rawValue, forK\

77 ey: UserDefaultsKeys.temperatureNotation)

78

79 let viewModel = SettingsViewTemperatureViewModel(temperature\

80 Notation: .celsius)

81

82 XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\

83 soryType.checkmark)

84 }

85

86 }

Press Command + U one more time to run the test suite of unit tests.

You can see the results in the Test Navigator or in the Report

Navigator.

Inspecting the Results In the Test Navigator

Inspecting the Results In the Report Navigator

In the next chapter, we write unit tests for the DayViewViewModel struct.

15 Using Stubs for Better Unit Tests

Testing the DayViewViewModel struct isn’t very different from testing the

view models of the SettingsViewController class. The only tricky aspect

is instantiating a DayViewViewModel instance in a unit test.

To instantiate a DayViewViewModel instance, we need a model. Should we

fetch weather data from the Dark Sky API during a test run? The answer

is a resounding “no”. To guarantee that the unit tests for the

DayViewViewModel struct are fast and reliable, we need stubs.

The idea is simple. We fetch a response from the Dark Sky API, save it in

the unit testing bundle, and load the response when we run the unit tests

for the view model. Let me show you how this works.

Adding Stub Data

I’ve already saved a response from the Dark Sky API to my desktop. This

is nothing more than a plain text file with JSON data. Before we can use

it in the test case, we add the file to the unit testing bundle. The JSON file

is included with the source files of this chapter. Drag it in the Stubs group

of the CloudyTests target.

Adding Stub Data

Make sure that Copy items if needed is checked and that the file is only

added to the CloudyTests target.

Adding Stub Data to CloudyTests Target

Adding Stub Data to CloudyTests Target

Loading Stub Data

Because we’ll use the stub data in multiple test cases, we first create a

helper method to load the stub data from the unit testing bundle. Create a

new file in the Extensions group of the unit testing bundle and name it

XCTestCase.swift.

Creating an Extension for XCTestCase

Replace the import statement for Foundation with an import statement

for XCTest and define an extensions for the XCTestCase class.

XCTestCase.swift

1 import XCTest

2

3 extension XCTestCase {

4

5 }

Name the helper method loadStubFromBundle(withName:extension:).

XCTestCase.swift

1 func loadStubFromBundle(withName name: String, extension: String) ->\

2 Data {

3

4 }

The method accepts two parameters:

the name of a file

the extension of a file

In loadStubFromBundle(withName:extension:), we fetch a reference to the

unit testing bundle, ask it for the URL of the file we’re interested in, and

use the URL to instantiate a Data instance.

XCTestCase.swift

1 func loadStubFromBundle(withName name: String, extension: String) ->\

2 Data {

3 let bundle = Bundle(for: classForCoder)

4 let url = bundle.url(forResource: name, withExtension: `extensio\

5 n`)

6

7 return try! Data(contentsOf: url!)

8 }

Notice that we force unwrap the url optional and, Heaven forbid, use the

try keyword with an exclamation mark. This is something I only ever do

when writing unit tests. You have to understand that we’re only interested

in the results of the unit tests. If anything else goes wrong, we made a

silly mistake, which we need to fix. In other words, I’m not interested in

error handling or safety when writing and running unit tests. If something

goes wrong, the unit tests fail anyway.

Unit Testing the Day View View Model

Unit Testing the Day View View Model

We can now create the test case for the DayViewViewModel struct. Create

a new test case and name the file DayViewViewModelTests.swift. We

start by adding an import statement for the Cloudy module. Don’t forget

to prefix the import statement with the testable attribute.

Creating DayViewViewModelTests.swift

DayViewViewModelTests.swift

1 import XCTest

2 @testable import Cloudy

3

4 class DayViewViewModelTests: XCTestCase {

5

6 // MARK: - Set Up & Tear Down

7

8 override func setUp() {

9 super.setUp()

10 }

11

12 override func tearDown() {

13 super.tearDown()

14 }

15

16 }

To simplify the unit tests, we won’t be instantiating a view model in each

of the unit tests. Instead, we create a view model, the view model we use

for testing, in the setUp() method. Let me show you how that works and

what the benefits are.

We first define a property for the view model. This means every unit test

will have access to a fully initialized view model, ready for testing.

will have access to a fully initialized view model, ready for testing.

DayViewViewModelTests.swift

1 import XCTest

2 @testable import Cloudy

3

4 class DayViewViewModelTests: XCTestCase {

5

6 // MARK: - Properties

7

8 var viewModel: DayViewViewModel!

9

10 // MARK: - Set Up & Tear Down

11

12 override func setUp() {

13 super.setUp()

14 }

15

16 override func tearDown() {

17 super.tearDown()

18 }

19

20 }

Notice that the type of the property is an implicitly unwrapped optional.

This is dangerous, but remember that we don’t care if the test suite

crashes and burns. If that happens, it means that we made a mistake we

need to fix. This is really important to understand. When we’re running

the unit tests, we’re interested in the test results. We very often use

shortcuts for convenience to improve the clarity and the readability of the

unit tests. This’ll become clear in a moment.

In the setUp() method, we invoke the

loadStubFromBundle(withName:extension:) helper method to load the

contents of the stub we added earlier and we use the Data object to

instantiate a WeatherData instance. The model is used to create the

DayViewViewModel instance we’re going to use in each of the unit tests.

DayViewViewModelTests.swift

1 override func setUp() {

2 super.setUp()

3

4 // Load Stub

5 let data = loadStubFromBundle(withName: "darksky", extension: "j\

6 son")

7 let weatherData: WeatherData = try! JSONDecoder.decode(data: dat\

8 a)

9

10 // Initialize View Model

11 viewModel = DayViewViewModel(weatherData: weatherData)

12 }

The first unit test is as simple as unit tests get. We test the date

computed property of the DayViewViewModel struct. We assert that the

value of the date computed property is equal to the value we expect.

DayViewViewModelTests.swift

1 // MARK: - Tests for Date

2

3 func testDate() {

4 XCTAssertEqual(viewModel.date, "Tue, July 11")

5 }

We can keep the unit test this simple because we control the stub data. If

we were to fetch a response from the Dark Sky API, we wouldn’t have a

clue what would come back. It would be slow, asynchronous, and prone

to all kinds of issues.

The second unit test we write is for the time computed property of the

DayViewViewModel struct. Because the value of the time computed

property depends on the user’s preference, stored in the user defaults

database, we have two unit tests to write.

DayViewViewModelTests.swift

1 // MARK: - Tests for Time

2

3 func testTime\_TwelveHour() {

4

5 }

6

7 func testTime\_TwentyFourHour() {

8

9 }

The body of the first unit test looks very similar to some of the unit tests

we wrote in the previous chapter. We set the time notation setting in the

user defaults database and assert that the value of the time computed

property is equal to the value we expect. Let me repeat that we can only

do this because we know the contents of the stub data and, as a result,

the model the view model manages.

DayViewViewModelTests.swift

1 func testTime\_TwelveHour() {

2 let timeNotation: TimeNotation = .twelveHour

3 UserDefaults.standard.set(timeNotation.rawValue, forKey: UserDef\

4 aultsKeys.timeNotation)

5

6 XCTAssertEqual(viewModel.time, "01:57 PM")

7 }

The second unit test for the time computed property is very similar. Only

the value we set in the user defaults database is different.

DayViewViewModelTests.swift

1 func testTime\_TwentyFourHour() {

2 let timeNotation: TimeNotation = .twentyFourHour

3 UserDefaults.standard.set(timeNotation.rawValue, forKey: UserDef\

4 aultsKeys.timeNotation)

5

6 XCTAssertEqual(viewModel.time, "13:57")

7 }

The remaining unit tests for the DayViewViewModel struct follow the same

pattern. Put the book aside and give them a try. I have to warn you,

though, the unit test for the image computed property is a bit trickier. But

you can do this. You can find the remaining unit tests below.

DayViewViewModelTests.swift

1 // MARK: - Tests for Summary

2

3 func testSummary() {

4 XCTAssertEqual(viewModel.summary, "Clear")

5 }

6

7 // MARK: - Tests for Temperature

8

9 func testTemperature\_Fahrenheit() {

10 let temperatureNotation: TemperatureNotation = .fahrenheit

11 UserDefaults.standard.set(temperatureNotation.rawValue, forKey: \

12 UserDefaultsKeys.temperatureNotation)

13

14 XCTAssertEqual(viewModel.temperature, "44.5 F")

15 }

16

17 func testTemperature\_Celsius() {

18 let temperatureNotation: TemperatureNotation = .celsius

19 UserDefaults.standard.set(temperatureNotation.rawValue, forKey: \

20 UserDefaultsKeys.temperatureNotation)

21

22 XCTAssertEqual(viewModel.temperature, "6.9 C")

23 }

24

25 // MARK: - Tests for Wind Speed

26

27 func testWindSpeed\_Imperial() {

28 let unitsNotation: UnitsNotation = .imperial

29 UserDefaults.standard.set(unitsNotation.rawValue, forKey: UserDe\

30 faultsKeys.unitsNotation)

31

32 XCTAssertEqual(viewModel.windSpeed, "6 MPH")

33 }

34

35 func testWindSpeed\_Metric() {

36 let unitsNotation: UnitsNotation = .metric

37 UserDefaults.standard.set(unitsNotation.rawValue, forKey: UserDe\

38 faultsKeys.unitsNotation)

39

40 print(viewModel.windSpeed)

41

42 XCTAssertEqual(viewModel.windSpeed, "10 KPH")

43 }

44

45 // MARK: - Tests for Image

46

47 func testImage() {

48 let viewModelImage = viewModel.image

49 let imageDataViewModel = UIImagePNGRepresentation(viewModelImage\

50 !)!

51 let imageDataReference = UIImagePNGRepresentation(UIImage(named:\

52 "clear-day")!)!

53

54 XCTAssertNotNil(viewModelImage)

55 XCTAssertEqual(viewModelImage!.size.width, 236.0)

56 XCTAssertEqual(viewModelImage!.size.height, 236.0)

57 XCTAssertEqual(imageDataViewModel, imageDataReference)

58 }

The unit test for the image computed property is slightly different.

Comparing images isn’t straightforward. We first make an assertion that

the value of the image computed property isn’t nil because it returns a

UIImage?.

DayViewViewModelTests.swift

1 XCTAssertNotNil(viewModelImage)

We then convert the image to a Data object and compare it to a reference

image, loaded from the application bundle. You can go as far as you like.

For example, I’ve also added assertions for the dimensions of the image.

This isn’t critical for this application, but it shows you what’s possible.

DayViewViewModelTests.swift

1 XCTAssertEqual(viewModelImage!.size.width, 236.0)

2 XCTAssertEqual(viewModelImage!.size.height, 236.0)

3 XCTAssertEqual(imageDataViewModel, imageDataReference)

Before we run the test suite, we need to tie up some loose ends. In the

tearDown() method, we reset the state we set in the unit tests.

DayViewViewModelTests.swift

1 override func tearDown() {

2 super.tearDown()

3

4 // Reset User Defaults

5 UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.time\

6 Notation)

7 UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.unit\

8 sNotation)

9 UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.temp\

10 eratureNotation)

11 }

Press Command + U to run the test suite to make sure the unit tests for

the DayViewViewModel struct pass.

Running the Test Suite

In the next chapter, we unit test the view models for the

WeekViewController class.

16 A Few More Unit Tests

Writing units tests for the view models of the WeekViewController class is

just as easy as writing unit tests for the DayViewViewModel struct. We start

with the unit tests for the WeekViewViewModel struct.

Unit Testing the Week View View Model

Create a new XCTestCase subclass and name the file

WeekViewViewModelTests.swift.

Creating WeekViewViewModelTests.swift

Add an import statement for the Cloudy module and define a property for

the view model like we did in the previous chapter. The approach we take

is identical to the approach we took in the previous chapter. The type of

the property is an implicitly unwrapped optional, WeekViewViewModel!.

WeekViewViewModelTests.swift

1 import XCTest

2 @testable import Cloudy

3

4 class WeekViewViewModelTests: XCTestCase {

5

6 // MARK: - Properties

7

8 var viewModel: WeekViewViewModel!

9

10 // MARK: - Set Up & Tear Down

11

12 override func setUp() {

13 super.setUp()

14 }

15

16 override func tearDown() {

17 super.tearDown()

18 }

19

20 }

In the setUp() method, we load the same stub from the unit testing

bundle, instantiate a WeatherData instance with it, and use the value of

the dailyData property to instantiate the view model. Remember that the

dailyData property is of type [WeatherDayData].

WeekViewViewModelTests.swift

1 override func setUp() {

2 super.setUp()

3

4 // Load Stub

5 let data = loadStubFromBundle(withName: "darksky", extension: "j\

6 son")

7 let weatherData: WeatherData = try! JSONDecoder.decode(data: dat\

8 a)

9

10 // Initialize View Model

11 viewModel = WeekViewViewModel(weatherData: weatherData.dailyData)

12 }

The unit tests for the WeekViewViewModel struct are very easy to write. The

simplest unit test of this book is the one for the numberOfSections

computed property since it always returns the value 1.

WeekViewViewModelTests.swift

1 // MARK: - Tests for Number of Sections

2

3 func testNumberOfSections() {

4 XCTAssertEqual(viewModel.numberOfSections, 1)

5 }

The unit test for numberOfDays is also easy to write. One assertion is all

we need.

WeekViewViewModelTests.swift

1 // MARK: - Tests for Number of Days

2

3 func testNumberOfDays() {

4 XCTAssertEqual(viewModel.numberOfDays, 8)

5 }

Testing the viewModel(for:) method is slightly more complicated. We can

take a few approaches. Remember that this method returns an object

that conforms to the WeatherDayRepresentable protocol. One approach is

to ask the view model for the object that corresponds with a particular

index and assert that the day and date properties are equal to the values

we expect based on the stub we added to the unit testing bundle.

WeekViewViewModelTests.swift

1 // MARK: - Tests for View Model for Index

2

3 func testViewModelForIndex() {

4 let weatherDayViewViewModel = viewModel.viewModel(for: 5)

5

6 XCTAssertEqual(weatherDayViewViewModel.day, "Saturday")

7 XCTAssertEqual(weatherDayViewViewModel.date, "July 15")

8 }

These are all the unit tests we need to write for the WeekViewViewModel

struct. Press Command + U to run the test suite.

Running the Test Suite

Unit Testing the Weather Day View View Model

Unit Testing the Weather Day View View Model

You should now be able to write the unit tests for the

WeatherDayViewViewModel struct. The unit tests are very similar to those of

the DayViewViewModel struct. The only difficulty is instantiating the view

model. Give it a try to see if you can make it work. You can find the

solution below.

We create a new file and name it

WeatherDayViewViewModelTests.swift.

Creating WeatherDayViewViewModelTests.swift

We add an import statement for the Cloudy module and define a

property for the view model of type WeatherDayViewViewModel!, an

implicitly unwrapped optional.

WeatherDayViewViewModelTests.swift

1 import XCTest

2 @testable import Cloudy

3

4 class WeatherDayViewViewModelTests: XCTestCase {

5

6 // MARK: - Properties

7

8 var viewModel: WeatherDayViewViewModel!

9

10 // MARK: - Set Up & Tear Down

11

12 override func setUp() {

13 super.setUp()

14 }

15

16 override func tearDown() {

17 super.tearDown()

18 }

19

20 }

We instantiate the view model in the setUp() method. We load the stub

from the unit testing bundle, create a WeatherData instance with it, and

use the WeatherData instance to initialize the view model. Because we

need a WeatherDayData instance to initialize the view model, we ask the

WeatherData instance for one. This is the only complexity of the unit tests

for the WeatherDayViewViewModel struct.

WeatherDayViewViewModelTests.swift

1 override func setUp() {

2 super.setUp()

3

4 // Load Stub

5 let data = loadStubFromBundle(withName: "darksky", extension: "j\

6 son")

7 let weatherData: WeatherData = try! JSONDecoder.decode(data: dat\

8 a)

9

10 // Initialize View Model

11 viewModel = WeatherDayViewViewModel(weatherDayData: weatherData.\

12 dailyData[5])

13 }

The unit tests should look familiar. They’re similar to the ones we wrote

for the DayViewViewModel struct.

WeatherDayViewViewModelTests.swift

1 // MARK: - Tests for Day

2

3 func testDay() {

4 XCTAssertEqual(viewModel.day, "Saturday")

5 }

6

7 // MARK: - Tests for Date

8

9 func testDate() {

10 XCTAssertEqual(viewModel.date, "July 15")

11 }

12

13 // MARK: - Tests for Temperature

14

15 func testTemperature\_Fahrenheit() {

16 let temperatureNotation: TemperatureNotation = .fahrenheit

17 UserDefaults.standard.set(temperatureNotation.rawValue, forKey: \

18 UserDefaultsKeys.temperatureNotation)

19

20 XCTAssertEqual(viewModel.temperature, "37 F - 47 F")

21 }

22

23 func testTemperature\_Celsius() {

24 let temperatureNotation: TemperatureNotation = .celsius

25 UserDefaults.standard.set(temperatureNotation.rawValue, forKey: \

26 UserDefaultsKeys.temperatureNotation)

27

28 XCTAssertEqual(viewModel.temperature, "3 C - 8 C")

29 }

30

31 // MARK: - Tests for Wind Speed

32

33 func testWindSpeed\_Imperial() {

34 let unitsNotation: UnitsNotation = .imperial

35 UserDefaults.standard.set(unitsNotation.rawValue, forKey: UserDe\

36 faultsKeys.unitsNotation)

37

38 XCTAssertEqual(viewModel.windSpeed, "1 MPH")

39 }

40

41 func testWindSpeed\_Metric() {

42 let unitsNotation: UnitsNotation = .metric

43 UserDefaults.standard.set(unitsNotation.rawValue, forKey: UserDe\

44 faultsKeys.unitsNotation)

45

46 XCTAssertEqual(viewModel.windSpeed, "2 KPH")

47 }

48

49 // MARK: - Tests for Image

50

51 func testImage() {

52 let viewModelImage = viewModel.image

53 let imageDataViewModel = UIImagePNGRepresentation(viewModelImage\

54 !)!

55 let imageDataReference = UIImagePNGRepresentation(UIImage(named:\

56 "cloudy")!)!

57

58 XCTAssertNotNil(viewModelImage)

59 XCTAssertEqual(viewModelImage!.size.width, 236.0)

60 XCTAssertEqual(viewModelImage!.size.height, 172.0)

61 XCTAssertEqual(imageDataViewModel, imageDataReference)

62 }

In the tearDown() method, we reset the state we set in the unit tests.

WeatherDayViewViewModelTests.swift

1 override func tearDown() {

2 super.tearDown()

3

4 // Reset User Defaults

5 UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.unit\

6 sNotation)

7 UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.temp\

8 eratureNotation)

9 }

Well done. We’ve now fully covered the view models with unit tests. Run

the test suite one more time to make sure all the unit tests pass.

the test suite one more time to make sure all the unit tests pass.

Running the Test Suite

In the second part of this book, we take the Model-View-ViewModel

pattern to the next level by introducing bindings.

17 Taking MVVM to the Next Level

You should now have a good understanding of what MVVM is and how it

can be used to cure some of the problems MVC suffers from. But we can

do better. Up until now, data in the application has flown in one direction.

The view controller asks the view model for data and populates the view

it manages. This is fine and many projects can greatly benefit from this

implementation of the Model-View-ViewModel pattern.

Data Flows in One Direction

But it’s time to show you how we can take the Model-View-ViewModel

pattern to the next level. In the next few chapters, we discuss how the

Model-View-ViewModel pattern can be used to not only populate a view

with data but also respond to changes made by the user or a change of

the environment.

One of Cloudy’s features is the ability for the user to search for and save

locations. When the user selects one of the saved locations, Cloudy

fetches weather data for that location and displays it to the user.

Let me show you how this works. When the user taps the location button

in the top left, the locations view controller is shown. It lists the current

location of the device as well as the user’s saved locations.

Cloudy can manage a list of locations.

To add a location to the list of saved locations, the user taps the plus

button in the top left. This brings up the add location view controller.

The user can add locations using the add location view

controller.

The user enters the name of the city she would like to add and Cloudy

uses the Core Location framework to forward geocode the name of that

city. Under the hood, Cloudy asks the Core Location framework for the

coordinates of the city the user has entered.

When the user enters a city in the search bar and taps the Search

button, the view controller uses a CLGeocder instance to forward geocode

the name of the city. Forward geocoding is an asynchronous operation.

The view controller updates the table view when the Core Location

framework returns the results of the geocoding request.

The current implementation of this feature uses the Model-View-

Controller pattern. But you want to know how we can improve this

dramatically using the Model-View-ViewModel pattern. That’s what you’re

here for.

How can we improve what we currently have? We’ll create a view model

responsible for everything related to responding to the user’s input. The

view model will send the geocoding request to Apple’s location services.

This is an asynchronous operation. When the geocoding request

completes, successfully or unsuccessfully, the view controller’s table view

is updated with the results of the geocoding request. This example will

show you how powerful the Model-View-ViewModel pattern can be when

it’s correctly implemented.

Data Flows in Both Directions

Let’s take a quick look at the current implementation, which uses the

Model-View-Controller pattern. We’re only going to focus on the

AddLocationViewController class.

The class defines outlets for a table view and a search bar. It also

maintains a list of Location instances. These are the results the Core

Location framework hands us. The Location type is a struct that makes

working with the results of the Core Location framework easier. The

CLGeocoder instance is responsible for making the geocoding requests.

Don’t worry if you’re not familiar with this class, it’s very easy to use.

AddLocationViewController.swift

1 class AddLocationViewController: UIViewController {

2

3 // MARK: - Properties

4

5 @IBOutlet var tableView: UITableView!

6 @IBOutlet var searchBar: UISearchBar!

7

8 // MARK: -

9

10 private var locations: [Location] = []

11

12 // MARK: -

13

14 private lazy var geocoder = CLGeocoder()

15

16 // MARK: -

17

18 var delegate: AddLocationViewControllerDelegate?

19

20 }

We also define a delegate protocol, AddLocationViewControllerDelegate,

to notify the LocationsViewController of the user’s selection. This isn’t

important for the rest of the discussion, though.

AddLocationViewController.swift

1 protocol AddLocationViewControllerDelegate {

2 func controller(\_ controller: AddLocationViewController, didAddL\

3 ocation location: Location)

4 }

The add location view controller is the delegate of the search bar and it

conforms to the UISearchBarDelegate protocol. When the user taps the

Search button, the text of the search bar is used as input for the

geocoding request.

AddLocationViewController.swift

1 extension AddLocationViewController: UISearchBarDelegate {

2

3 func searchBarSearchButtonClicked(\_ searchBar: UISearchBar) {

4 // Hide Keyboard

5 searchBar.resignFirstResponder()

6

7 // Forward Geocode Address String

8 geocode(addressString: searchBar.text)

9 }

10

11 func searchBarCancelButtonClicked(\_ searchBar: UISearchBar) {

12 // Hide Keyboard

13 searchBar.resignFirstResponder()

14

15 // Clear Locations

16 locations = []

17

18 // Update Table View

19 tableView.reloadData()

20 }

21

22 }

Notice that we already use a dash of MVVM in the

UITableViewDataSource protocol to populate the table view. We take it a

few steps further in the next few chapters.

AddLocationViewController.swift

1 extension AddLocationViewController: UITableViewDataSource {

2

3 func tableView(\_ tableView: UITableView, numberOfRowsInSection s\

4 ection: Int) -> Int {

5 return locations.count

6 }

7

8 func tableView(\_ tableView: UITableView, cellForRowAt indexPath:\

9 IndexPath) -> UITableViewCell {

10 guard let cell = tableView.dequeueReusableCell(withIdentifie\

11 r: LocationTableViewCell.reuseIdentifier, for: indexPath) as? Locati\

12 onTableViewCell else { fatalError("Unexpected Table View Cell") }

13

14 // Fetch Location

15 let location = locations[indexPath.row]

16

17 // Create View Model

18 let viewModel = LocationsViewLocationViewModel(location: loc\

19 ation.location, locationAsString: location.name)

20

21 // Configure Table View Cell

22 cell.configure(withViewModel: viewModel)

23

24 return cell

25 }

26

27 }

The Core Location framework makes forward geocoding easy. The magic

happens in the geocode(addressString:) method.

AddLocationViewController.swift

1 private func geocode(addressString: String?) {

2 guard let addressString = addressString else {

3 // Clear Locations

4 locations = []

5

6 // Update Table View

7 tableView.reloadData()

8

9 return

10 }

11

12 // Geocode City

13 geocoder.geocodeAddressString(addressString) { [weak self] (plac\

14 emarks, error) in

15 DispatchQueue.main.async {

16 // Process Forward Geocoding Response

17 self?.processResponse(withPlacemarks: placemarks, error:\

18 error)

19 }

20 }

21 }

If the user’s input is empty, we clear the table view. If the user enters a

valid location, we invoke geocodeAddressString(\_:completionHandler:)

on the CLGeocoder instance. Core Location hands us an array of

CLPlacemark instances if the geocoding request is successful. The

CLPlacemark class is defined in the Core Location framework and is used

to store the metadata for a geographic location.

The response of the geocoding request is handled in the

processResponse(withPlacemarks:error:) method. We create an array of

Location instances from the array of CLPlacemark instances and update

the table view. That’s it.

AddLocationViewController.swift

1 private func processResponse(withPlacemarks placemarks: [CLPlacemark\

2 ]?, error: Error?) {

3 if let error = error {

4 print("Unable to Forward Geocode Address (\(error))")

5

6 } else if let matches = placemarks {

7 // Update Locations

8 locations = matches.flatMap({ (match) -> Location? in

9 guard let name = match.name else { return nil }

10 guard let location = match.location else { return nil }

11 return Location(name: name, latitude: location.coordinat\

12 e.latitude, longitude: location.coordinate.longitude)

13 })

14

15 // Update Table View

16 tableView.reloadData()

17 }

18 }

We covered the most important details of the AddLocationViewController

class. In the next chapter, I show you what we’re up to. How can we lift

the view controller from some of its responsibilities?

18 What Are the Options

The question we need to answer in the next few chapters is “How can we

tie everything together?” This is a problem many developers new to the

Model-View-ViewModel pattern struggle with. It’s not difficult to use the

Model-View-ViewModel pattern to push data from the controller layer to

the view layer. We’ve already covered that extensively in this book. But

how do you respond to user interactions or changes of the environment,

and update the user interface … automatically?

Remember that I told you that the Model-View-ViewModel pattern

originated on the .NET platform. One of the reasons it works so well on

the .NET platform is because of bindings. Because of this, the Model-

View-ViewModel pattern is sometimes referred to as the Model-View-

Binder pattern. It ensures that the view layer and the model layer are

synchronized.

What we want to avoid is having to write glue code. We don’t want to

manually push changes from the view model to the view controller’s view.

That’s the task of the view controller … but … that’s exactly what we

want to avoid. Right? We want a better, more robust solution.

Unfortunately, Swift currently doesn’t have bindings. You can roll your

own implementation using closures or key-value-observing. I like to refer

to this as “DIY bindings”, “Do It Yourself bindings”. That’s the solution we

start with. But this solution isn’t scalable and it isn’t terribly elegant.

Complex applications need a better solution. There are several options

available, such as Bond, RxSwift, and ReactiveCocoa. The solution we’ll

be using in this book is RxSwift. It’s the most popular option and the one

I have come to appreciate most. It’s also easy to pick up.

I want to emphasize that you don’t need to be familiar with RxSwift to

understand the next few chapters. The emphasis of the following

chapters lies on the implementation of the Model-View-ViewModel

pattern, not understanding RxSwift. This also means that I won’t explain

pattern, not understanding RxSwift. This also means that I won’t explain

the details that relate to RxSwift in great detail. You can use any bindings

solution you like, including a custom one.

19 DIY Bindings

Before we use RxSwift to implement the Model-View-ViewModel pattern,

I want to show you how you can use closures to implement a custom

solution. I usually refer to this solution as “DIY bindings” or “Do It Yourself

bindings”.

Creating the View Model

It’s time to write some code. If you’d like to follow along, open the starter

project of this chapter. By now, you should know how to implement a

view model. Create a new group, View Models, in the Add Location

View Controller group, and add a new Swift file to that group,

AddLocationViewViewModel.swift.

Creating AddLocationViewViewModel.swift

We first define the AddLocationViewViewModel class. The implementation

isn’t too difficult, but there are several important details to pay attention

to.

AddLocationViewViewModel.swift

1 import Foundation

2

3 class AddLocationViewViewModel {

4

5 }

Implementing the View Model

We import the Core Location framework and define a private, lazy,

variable property, geocoder, of type CLGeocoder. The add location view

controller no longer needs to know about the Core Location framework

and it shouldn’t perform the geocoding requests. This is now handled by

the view model.

AddLocationViewViewModel.swift

1 import Foundation

2 import CoreLocation

3

4 class AddLocationViewViewModel {

5

6 // MARK: - Properties

7

8 private lazy var geocoder = CLGeocoder()

9

10 }

We also define a variable property, query, of type String. This property is

updated by the view controller whenever the user taps the search or

cancel button.

AddLocationViewViewModel.swift

1 import Foundation

2 import CoreLocation

3

4 class AddLocationViewViewModel {

5

6 // MARK: - Properties

7

8 var query: String = ""

9

10 // MARK: -

11

12 private lazy var geocoder = CLGeocoder()

13

14 }

The view model needs to keep track of the locations and whether a

geocoding request is in progress. This means we need to declare a few

variables to manage state. We define querying, a boolean that knows

whether a geocoding request is in progress, and locations, an array of

Location instances. The locations property stores the result of the

geocoding request.

AddLocationViewViewModel.swift

1 import Foundation

2 import CoreLocation

3

4 class AddLocationViewViewModel {

5

6 // MARK: - Properties

7

8 var query: String = ""

9

10

11 // MARK: -

12

13 private var querying: Bool = false

14 private var locations: [Location] = []

15

16 // MARK: -

17

18 private lazy var geocoder = CLGeocoder()

19

20 }

Notice that both properties are declared private. The view controller

doesn’t need to know about these properties and it certainly shouldn’t be

able to modify the values of these properties. Instead, we define

properties and methods the view controller can use to ask the view model

for the data it needs to update its view.

We define two computed properties, numberOfLocations and

hasLocations. The numberOfLocations computed property returns the

number of Location instances stored in the private locations property.

The hasLocations computed property returns true if numberOfLocations is

greater than 0.

AddLocationViewViewModel.swift

1 import Foundation

2 import CoreLocation

3

4 class AddLocationViewViewModel {

5

6 // MARK: - Properties

7

8 var query: String = ""

9

10

11 // MARK: -

12

13 private var querying: Bool = false

14 private var locations: [Location] = []

15

16 // MARK: -

17

18 var hasLocations: Bool { return numberOfLocations > 0 }

19 var numberOfLocations: Int { return locations.count }

20

21 // MARK: -

22

23 private lazy var geocoder = CLGeocoder()

24

25 }

We also define two convenience methods to make the view controller’s

task very easy. The first method, location(at:), returns the location for a

particular index. The method returns nil if the value of index is greater

than or equal to the number of locations stored in the locations property.

AddLocationViewViewModel.swift

1 func location(at index: Int) -> Location? {

2 guard index < locations.count else { return nil }

3 return locations[index]

4 }

The second method, viewModelForLocation(at:), goes one step further

and returns an object of type LocationRepresentable?. It uses the

location(at:) method to fetch the Location instance that corresponds

with the value of index. The Location instance is used to create an

instance of the LocationsViewLocationViewModel struct.

AddLocationViewViewModel.swift

1 func viewModelForLocation(at index: Int) -> LocationRepresentable? {

2 guard let location = location(at: index) else { return nil }

3 return LocationsViewLocationViewModel(location: location.locatio\

4 n, locationAsString: location.name)

5 }

These properties and methods will make the implementations of the

UITableViewDataSource and UITableViewDelegate protocols in the

AddLocationViewController class trivial.

Performing Geocoding Requests

When the value of the query property changes, the view model should

perform a geocoding request. We can accomplish this using key-valueobserving

or by implementing a property observer. Let’s keep it simple

and implement a property observer.

In the property observer, we invoke the geocode(addressString:) method,

passing in the value of the query property.

AddLocationViewViewModel.swift

1 var query: String = "" {

2 didSet {

3 geocode(addressString: query)

4 }

5 }

The implementation of the geocode(addressString:) method is

straightforward. We use a guard statement to make sure addressString

isn’t equal to nil and the string stored in addressString isn’t empty. We

set locations to an empty array if either of these conditions isn’t met.

AddLocationViewViewModel.swift

1 // MARK: - Helper Methods

2

3 private func geocode(addressString: String?) {

4 guard let addressString = addressString, !addressString.isEmpty \

5 else {

6 locations = []

7 return

8 }

9 }

If addressString has a valid value, we set querying to true to indicate a

geocoding request is in flight. We then invoke

geocodeAddressString(\_:completionHandler:) on the CLGeocoder

instance.

AddLocationViewViewModel.swift

1 // MARK: - Helper Methods

2

3 private func geocode(addressString: String?) {

4 guard let addressString = addressString, !addressString.isEmpty \

5 else {

6 locations = []

7 return

8 }

9

10 querying = true

11

12 // Geocode Address String

13 geocoder.geocodeAddressString(addressString) { [weak self] (plac\

14 emarks, error) in

15

16 }

17 }

In the completion handler, we declare a helper variable, locations, of

type [Location] and we set querying to false to indicate the geocoding

request has completed, successfully or unsuccessfully. If an error was

thrown, the error is printed to the console.

AddLocationViewViewModel.swift

1 // MARK: - Helper Methods

2

3 private func geocode(addressString: String?) {

4 guard let addressString = addressString, !addressString.isEmpty \

5 else {

6 locations = []

7 return

8 }

9

10 querying = true

11

12 // Geocode Address String

13 geocoder.geocodeAddressString(addressString) { [weak self] (plac\

14 emarks, error) in

15 var locations: [Location] = []

16

17 self?.querying = false

18

19 if let error = error {

20 print("Unable to Forward Geocode Address (\(error))")

21

22 } else {

23

24 }

25 }

26 }

If an array with CLPlacemark instances is returned, we use flatMap(\_:) to

convert the placemarks to Location instances. We only create a Location

instance if the placemark has a name and a set of coordinates. The

name and coordinates of the placemark are used to create a Location

instance.

Before we return from the completion handler, the locations property is

updated with the value of the locations variable.

AddLocationViewViewModel.swift

1 // MARK: - Helper Methods

2

3 private func geocode(addressString: String?) {

4 guard let addressString = addressString, !addressString.isEmpty \

5 else {

6 locations = []

7 return

8 }

9

10 querying = true

11

12 // Geocode Address String

13 geocoder.geocodeAddressString(addressString) { [weak self] (plac\

14 emarks, error) in

15 var locations: [Location] = []

16

17 self?.querying = false

18

19 if let error = error {

20 print("Unable to Forward Geocode Address (\(error))")

21

22 } else if let \_placemarks = placemarks {

23 locations = \_placemarks.flatMap({ (placemark) -> Locatio\

24 n? in

25 guard let name = placemark.name else { return nil }

26 guard let location = placemark.location else { retur\

27 n nil }

28 return Location(name: name, latitude: location.coord\

29 inate.latitude, longitude: location.coordinate.longitude)

30 })

31 }

32

33 self?.locations = locations

34 }

35 }

Notifying the View Controller

You may be wondering how the view controller is notified when the

values of querying and locations have changed. That’s where closures

come into play. We need to define two more properties. Both properties

are closures.

The first property, queryingDidChange, is a closure that accepts a boolean

as its only argument. The second property, locationsDidChange, is a

closure that accepts an array of Location instances as its only argument.

Both properties have an optional type.

AddLocationViewViewModel.swift

1 // MARK: -

2

3 var queryingDidChange: ((Bool) -> ())?

4 var locationsDidChange: (([Location]) -> ())?

After the view controller sets the values of these properties, as we’ll see

in a moment, it is automatically notified when the values of querying or

locations have changed. The last piece of the puzzle is invoking these

closures at the appropriate time.

We have a few options. We can use key-value-observing or, as we did

earlier, implement a property observer. Even though the latter is the

easiest, I hope you can see that this isn’t a scalable solution for large or

complex applications.

Whenever the querying property is set, the closure stored in

queryingDidChange is invoked and the value of the querying property is

passed in as an argument.

AddLocationViewViewModel.swift

1 private var querying: Bool = false {

2 didSet {

3 queryingDidChange?(querying)

4 }

5 }

We implement a similar property observer for the locations property.

Whenever the locations property is set, the closure stored in

locationsDidChange is invoked and the value of the locations property is

passed in as an argument.

AddLocationViewViewModel.swift

1 private var locations: [Location] = [] {

2 didSet {

3 locationsDidChange?(locations)

4 }

5 }

Refactoring the Add Location View Controller

It’s time to refactor the AddLocationViewController class. We start by

removing any references to the Core Location framework, including the

geocoder property, the geocode(addressString:) method, and the

processResponse(withPlacemarks:error:) method.

We can also remove the locations property. The view controller no

longer stores the results of the geocoding request. That’s now the

responsibility of the view model.

We need to declare two properties. The first property is an outlet for a

UIActivityIndicatorView, which we’ll add to the storyboard in a moment.

AddLocationViewController.swift

1 @IBOutlet var activityIndicatorView: UIActivityIndicatorView!

The second property is for the view model. The property is named

viewModel and is of type AddLocationViewViewModel!. Notice that the

property is an implicitly unwrapped optional.

AddLocationViewController.swift

1 var viewModel: AddLocationViewViewModel!

We initialize the view model in the viewDidLoad() method of the

AddLocationViewController class. We could also initialize the view model

in the prepare(for:sender:) method of the LocationsViewController

class and inject it into the add location view controller. That’s a technique

I prefer. This is especially useful if the view model has dependencies of

its own.

Later in the book, however, it’s important that we initialize the view model

in the add location view controller. Don’t worry about this for now. It’ll

become clear in a later chapter.

LocationsViewController.swift

1 \*\*AddLocationViewController.swift\*\*

2

3 ```swift

4 override func viewDidLoad() {

5 super.viewDidLoad()

6

7 // Set Title

8 title = "Add Location"

9

10 // Initialize View Model

11 viewModel = AddLocationViewViewModel()

12 }

In the viewDidLoad() method of the AddLocationViewController class, we

assign values to the locationsDidChange and queryingDidChange

properties.

AddLocationViewController.swift

1 override func viewDidLoad() {

2 super.viewDidLoad()

3

4 // Set Title

5 title = "Add Location"

6

7 // Initialize View Model

8 viewModel = AddLocationViewViewModel()

9

10 // Configure View Model

11 viewModel.locationsDidChange = { [unowned self] (locations) in

12 self.tableView.reloadData()

13 }

14

15 viewModel.queryingDidChange = { [unowned self] (querying) in

16 if querying {

17 self.activityIndicatorView.startAnimating()

18 } else {

19 self.activityIndicatorView.stopAnimating()

20 }

21 }

22 }

Let me repeat how this works. When the value of the private locations

property of the view model changes, the locationsDidChange closure is

invoked. In response, the view controller updates the table view by calling

reloadData() on the table view.

The same applies to the queryingDidChange property. If the value of

querying is true, we invoke startAnimating() on the activity indicator

view. If the value of querying is false we invoke stopAnimating() on the

activity indicator view.

Updating the User Interface

Remember that we need to add an activity indicator view to the main

storyboard. Open Main.storyboard and locate the Add Location View

Controller Scene. Add an activity indicator view from the Object Library

and add the necessary layout constraints. Connect the activity indicator

view to the outlet we created a moment ago.

Updating the User Interface

With the activity indicator view selected, open the Attributes Inspector

on the right and check Hides When Stopped. The Hidden checkbox

should automatically be checked as a result.

Configuring the Activity Indicator View

More Refactoring

Before we run the application, we need to update the implementations of

the UITableViewDataSource, UITableViewDelegate, and

UISearchBarDelegate protocols.

UITableViewDataSource Protocol

UITableViewDataSource Protocol

Updating the UITableViewDataSource protocol is very easy thanks to the

view model. We can even delete a few lines of code in the

tableView(\_:cellForRowAt:) method.

AddLocationViewController.swift

1 extension AddLocationViewController: UITableViewDataSource {

2

3 func tableView(\_ tableView: UITableView, numberOfRowsInSection s\

4 ection: Int) -> Int {

5 return viewModel.numberOfLocations

6 }

7

8 func tableView(\_ tableView: UITableView, cellForRowAt indexPath:\

9 IndexPath) -> UITableViewCell {

10 guard let cell = tableView.dequeueReusableCell(withIdentifie\

11 r: LocationTableViewCell.reuseIdentifier, for: indexPath) as? Locati\

12 onTableViewCell else { fatalError("Unexpected Table View Cell") }

13

14 if let viewModel = viewModel.viewModelForLocation(at: indexP\

15 ath.row) {

16 // Configure Table View Cell

17 cell.configure(withViewModel: viewModel)

18 }

19

20 return cell

21 }

22

23 }

UITableViewDelegate Protocol

In the tableView(\_:didSelectRowAt:) method, we need to rewrite one

line. We ask the view model for the Location instance that corresponds

with the row the user tapped.

AddLocationViewController.swift

1 extension AddLocationViewController: UITableViewDelegate {

2

3 func tableView(\_ tableView: UITableView, didSelectRowAt indexPat\

4 h: IndexPath) {

5 guard let location = viewModel.location(at: indexPath.row) e\

6 lse { return }

7

8 // Notify Delegate

9 delegate?.controller(self, didAddLocation: location)

10

11 // Pop View Controller From Navigation Stack

12 navigationController?.popViewController(animated: true)

13 }

14

15 }

I hope it’s clear that the heavy lifting is done by the view model. The view

controller is only tasked with populating the view it manages and

responding to user interaction.

UISearchBarDelegate Protocol

In searchBarSearchButtonClicked(\_:) and

searchBarCancelButtonClicked(\_:), we set the query property of the view

model. That in turn triggers a geocoding request in the view model.

AddLocationViewController.swift

1 extension AddLocationViewController: UISearchBarDelegate {

2

3 func searchBarSearchButtonClicked(\_ searchBar: UISearchBar) {

4 // Hide Keyboard

5 searchBar.resignFirstResponder()

6

7 // Forward Geocode Address String

8 viewModel.query = searchBar.text ?? ""

9 }

10

11 func searchBarCancelButtonClicked(\_ searchBar: UISearchBar) {

12 // Hide Keyboard

13 searchBar.resignFirstResponder()

14

15 // Forward Geocode Address String

16 viewModel.query = searchBar.text ?? ""

17 }

18

19 }

It’s time to run the application to see if everything is still working. We

should now also see an activity indicator view when a geocoding request

is in flight.

This looks good. At the moment, we’re using a primitive bindings solution.

This is ideal to illustrate how the Model-View-ViewModel pattern plays

together with bindings under the hood. It takes away the magic that you

see when you start using a more advanced solution, such as Bond,

RxSwift, or ReactiveCocoa.

But since we’re taking the Model-View-ViewModel pattern to the next

level, we need to step up our game and look for a better bindings

solution. In the next few chapters, we refactor the current implementation

to use RxSWift and RxCocoa.

20 Why RxSwift

Before we refactor the AddLocationViewViewModel class, I’d like to take a

few minutes to explain my motivation for using RxSwift and RxCocoa.

There are several reasons.

It’s a Library

RxSwift is a reactive extension for the Swift language. ReactiveX has

been gaining in popularity ever since Rx.NET was open sourced several

years ago. Reactive extensions are available for many languages,

including Java, C#, JavaScript, and Python.

What I like about ReactiveX is that it doesn’t force you to adopt a

particular architecture. You can use Rx without using the Model-View-

ViewModel pattern. You should see Rx as a tool, not an architecture.

Testability

RxSwift has been around for quite a while and it’s a robust

implementation of the ReactiveX API. It’s fantastic to use and the more

you use it the more you experience the power and versatility of

ReactiveX.

The RxSwift team has invested heavily in testing and making it easy to

test reactive code. In a later chapter, we test the reactive code we write

and that’ll show you how important this is.

Powerful

While RxSwift has no relation to the Cocoa APIs, there is an extension

for Cocoa, RxCocoa. Most of the UIKit components you use day in day

out are reactified. In the next chapters, you find out how this works.

The combination of RxSwift and RxCocoa is amazing. If you’re new to

RxSwift and RxCocoa, then I encourage you to give it a try. I hope the

next few chapters can convince you to take a closer look at RxSwift and

next few chapters can convince you to take a closer look at RxSwift and

RxCocoa.

21 Integrating RxSwift and RxCocoa

This chapter hasn’t been updated for Xcode 9 and Swift 4. This

chapter will be updated as soon as RxSwift/RxCocoa officially

support Swift 4.

There are several options to integrate RxSwift and RxCocoa into a

project. The README of the RxSwift project shows you the different

possibilities. I mostly use CocoaPods and that’s the approach I take in

this book. If you’d like to follow along with me, make sure you have

CocoaPods installed. You can find more information about installing

CocoaPods on the CocoaPods website.

Defining Dependencies

Open Terminal, navigate to the root of the project, and execute the pod

init command to have CocoaPods create a Podfile for the project. Open

the Podfile in a text editor and add RxSwift and RxCocoa as

dependencies of the Cloudy target.

I’m also going to add RxTest and RxBlocking as dependencies of the

CloudyTests target. These libraries are also part of the RxSwift project

and are very helpful for testing reactive code. This is what the project’s

Podfile should look like.

1 target 'Cloudy' do

2 platform :ios, '10.0'

3 use\_frameworks!

4

5 pod 'RxSwift'

6 pod 'RxCocoa'

7

8 target 'CloudyTests' do

9 inherit! :search\_paths

10

11 pod 'RxTest'

12 pod 'RxBlocking'

13 end

14 end

Installing Dependencies

At the time of writing, RxSwift 3.5.0 is the latest release. Execute pod

install to install the dependencies listed in the project’s Podfile.

CocoaPods automatically creates a workspace for us, which we need to

use from now on.

1 Analyzing dependencies

2 Downloading dependencies

3 Installing RxBlocking (3.5.0)

4 Installing RxCocoa (3.5.0)

5 Installing RxSwift (3.5.0)

6 Installing RxTest (3.5.0)

7 Generating Pods project

8 Integrating client project

9

10 [!] Please close any current Xcode sessions and use `Cloudy.xcworksp\

11 ace` for this project from now on.

12 Sending stats

13 Pod installation complete! There are 4 dependencies from the Podfile\

14 and 4 total pods installed.

Open the workspace CocoaPods has created for us and build the project

to make sure everything is working as expected. You should see no

warnings or errors. Ready? It’s time to refactor the

AddLocationViewViewModel class.

22 Refactoring the View Model

This chapter hasn’t been updated for Xcode 9 and Swift 4. This

chapter will be updated as soon as RxSwift/RxCocoa officially

support Swift 4.

Refactoring the View Model

Make sure you open the workspace CocoaPods created for us in the

previous chapter. Open AddLocationViewViewModel.swift and add an

import statement for RxSwift and RxCocoa at the top.

AddLocationViewViewModel.swift

1 import RxSwift

2 import RxCocoa

3 import Foundation

4 import CoreLocation

5

6 class AddLocationViewViewModel {

7

8 ...

9

10 }

The search bar of the add location view controller drives the view model.

At the moment, the view controller sets the value of the query property

every time the text of the search bar changes. But we can do better. We

can replace the query property and pass a driver of type String to the

initializer of the view model. Let me show you how this works.

We define an initializer for the AddLocationViewViewModel class. The

initializer accepts one argument, a driver of type String. Don’t worry if

you’re not familiar with drivers. Think of a driver as a stream or sequence

of values. Instead of having a property, query, with a value, a driver is a

stream or sequence of values other objects can subscribe to. That’s all

you need to know about drivers to follow along.

AddLocationViewViewModel.swift

1 // MARK: - Initializtion

2

3 init(query: Driver<String>) {

4

5 }

Since we’re using RxSwift and RxCocoa, it’d be crazy not to take

advantage of the other features these libraries offer. In the initializer, we

apply two operators to the query driver. We apply the throttle(\_:)

operator, to limit the number of requests that are sent in a period of time,

and the distinctUntilChanged() operator, to prevent sending geocoding

requests to Apple’s location services for the same query.

AddLocationViewViewModel.swift

1 // MARK: - Initializtion

2

3 init(query: Driver<String>) {

4 query

5 .throttle(0.5)

6 .distinctUntilChanged()

7 }

We subscribe to the sequence of values by invoking the

drive(onNext:onCompleted:onDisposed:) method on the sequence. The

onNext handler is invoked when a new value is emitted by the sequence.

This happens when the user modifies the text in the search bar.

AddLocationViewViewModel.swift

1 // MARK: - Initializtion

2

3 init(query: Driver<String>) {

4 query

5 .throttle(0.5)

6 .distinctUntilChanged()

7 .drive(onNext: { [weak self] (addressString) in

8 self?.geocode(addressString: addressString)

9 })

10 .disposed(by: disposeBag)

11 }

When a new value is emitted by the sequence, we send a geocoding

request by invoking the geocode(addressString:) method, which we

implemented earlier in this book. You don’t need to worry about the

disposed(by:) method call. It is related to memory management and is

specific to RxSwift. To make this work, we need to define the disposeBag

property in the AddLocationViewViewModel class.

AddLocationViewViewModel.swift

1 // MARK: -

2

3 private let disposeBag = DisposeBag()

Reducing State

The current implementation of the AddLocationViewViewModel class keeps

a reference to the results of the geocoding requests. In other words, it

manages state. While this isn’t a problem, the fewer bits of state an

object keeps the better. This is another advantage of reactive

programming. Let me show you what I mean.

We can improve this with RxSwift and RxCocoa by keeping a reference

to the stream of results of the geocoding requests. The result is that the

view model no longer manages state, it simply holds a reference to the

pipeline through which the results of the geocoding requests flow.

The change is small, but there are several details that need our attention.

We declare a constant, private property \_locations of type Variable. The

Variable is of type [Location]. You can think of a Variable as the

pipeline and [Location] as the data that flows through that pipeline. We

initialize the pipeline with an empty array of locations.

AddLocationViewViewModel.swift

1 private let \_locations = Variable<[Location]>([])

We can do the same for the querying property. We declare a constant,

private property, \_querying, of type Variable. The Variable is of type

Bool. This means that the values that flow through the pipeline are

boolean values.

AddLocationViewViewModel.swift

1 private let \_querying = Variable<Bool>(false)

There’s a good reason for declaring these properties private. What we

expose to the view controller are drivers. What’s the difference between

drivers and variables? To keep it simple, think of drivers as read-only and

variables as read-write. We don’t want the view controller to make

changes to the stream of locations, for example. The drivers we expose

to the view controller are querying and locations.

AddLocationViewViewModel.swift

1 var querying: Driver<Bool> { return \_querying.asDriver() }

2 var locations: Driver<[Location]> { return \_locations.asDriver() }

The syntax may look daunting, but it really isn’t. querying is a computed

property of type Driver. The driver is of type Bool. The implementation is

simple. We return the Variable \_querying as a driver. The same is true

for locations. locations is a computed property of type Driver. The

driver is of type [Location]. We return the Variable \_locations as a

driver.

We expose two computed properties and we simply return the private

variables as drivers. Are you still with me?

Let’s clean up the pieces we no longer need. We can remove a few

properties:

the locations property

the old query property

the old querying property

And while we’re at it, we no longer need:

the queryingDidChange property

the locationsDidChange property

Great. The last thing we need to do is make a few changes to how the

view model accesses the array of locations. The changes are minor. A

reactive Variable exposes its current value through its value property.

This means we need to update:

the numberOfLocations computed property

the location(at:) method

the geocode(addressString:) method

AddLocationViewViewModel.swift

1 var numberOfLocations: Int { return \_locations.value.count }

AddLocationViewViewModel.swift

1 func location(at index: Int) -> Location? {

2 guard index < \_locations.value.count else { return nil }

3 return \_locations.value[index]

4 }

In geocode(addressString:), we also need to replace querying with

\_querying.value. We access the current value of the \_querying reactive

Variable through its value property.

AddLocationViewViewModel.swift

1 private func geocode(addressString: String?) {

2 guard let addressString = addressString, !addressString.isEmpty \

3 else {

4 \_locations.value = []

5 return

6 }

7

8 \_querying.value = true

9

10 // Geocode Address String

11 geocoder.geocodeAddressString(addressString) { [weak self] (plac\

12 emarks, error) in

13 var locations: [Location] = []

14

15 self?.\_querying.value = false

16

17 if let error = error {

18 print("Unable to Forward Geocode Address (\(error))")

19

20 } else if let \_placemarks = placemarks {

21 locations = \_placemarks.flatMap({ (placemark) -> Locatio\

22 n? in

23 guard let name = placemark.name else { return nil }

24 guard let location = placemark.location else { retur\

25 n nil }

26 return Location(name: name, latitude: location.coord\

27 inate.latitude, longitude: location.coordinate.longitude)

28 })

29 }

30

31 self?.\_locations.value = locations

32 }

33 }

That looks good. We can’t run the application yet because we’ve made

some breaking changes. We need to make a few modifications to the

AddLocationViewController class.

23 Refactoring the View Controller

Open AddLocationViewController.swift and add an import statement

for RxSwift and RxCocoa at the top.

AddLocationViewController.swift

1 import UIKit

2 import RxSwift

3 import RxCocoa

4

5 protocol AddLocationViewControllerDelegate {

6 func controller(\_ controller: AddLocationViewController, didAddL\

7 ocation location: Location)

8 }

9

10 class AddLocationViewController: UIViewController {

11

12 ...

13

14 }

We also need to declare a property, disposeBag, of type DisposeBag. As I

mentioned in the previous chapter, don’t worry about this if you’re not

familiar with RxSwift. The goal is to learn how the Model-View-ViewModel

pattern works with bindings. We’re not here to learn RxSwift.

AddLocationViewController.swift

1 // MARK: -

2

3 private let disposeBag = DisposeBag()

Our next stop is the viewDidLoad() method of the

AddLocationViewController class. We need to update the initializer of the

AddLocationViewViewModel class. We pass a driver as the only argument

of the initializer. Because we imported RxCocoa, we have access to the

reactive extensions of UISearchBar.

AddLocationViewController.swift

1 // MARK: - View Life Cycle

2

3 override func viewDidLoad() {

4 super.viewDidLoad()

5

6 // Set Title

7 title = "Add Location"

8

9 // Initialize View Model

10 viewModel = AddLocationViewViewModel(query: searchBar.rx.text.or\

11 Empty.asDriver())

12

13 // Configure View Model

14 viewModel.locationsDidChange = { [unowned self] (locations) in

15 self.tableView.reloadData()

16 }

17

18 viewModel.queryingDidChange = { [unowned self] (querying) in

19 if querying {

20 self.activityIndicatorView.startAnimating()

21 } else {

22 self.activityIndicatorView.stopAnimating()

23 }

24 }

25 }

A search bar emits a sequence of String values. We ask it for a

reference to that sequence. The orEmpty operator converts any nil

values to an empty string. The asDriver() method turns the sequence

into a driver. We pass this driver of type String to the initializer of the

AddLocationViewViewModel class.

AddLocationViewController.swift

1 // Initialize View Model

2 viewModel = AddLocationViewViewModel(query: searchBar.rx.text.orEmpt\

3 y.asDriver())

We can remove the remaining lines from the viewDidLoad() method.

Instead, we’re going to use bindings to update the user interface if the

view model performs a geocoding request and when it receives a

response.

AddLocationViewController.swift

1 // MARK: - View Life Cycle

2

3 override func viewDidLoad() {

4 super.viewDidLoad()

5

6 // Set Title

7 title = "Add Location"

8

9 // Initialize View Model

10 viewModel = AddLocationViewViewModel(query: searchBar.rx.text.or\

11 Empty.asDriver())

12 }

We listen for events of the locations driver of the view model. If a new

event is emitted, the table view is reloaded. Because the view model is

owned by the view controller, we use an unowned reference to self

within the closure.

AddLocationViewController.swift

1 // Drive Table View

2 viewModel.locations.drive(onNext: { [unowned self] (\_) in

3 // Update Table View

4 self.tableView.reloadData()

5 })

6 .disposed(by: disposeBag)

To show you how powerful and elegant Rx is, we use the querying driver

of the view model to start and stop animating the activity indicator view.

AddLocationViewController.swift

1 // Drive Activity Indicator View

2 viewModel.querying.drive(activityIndicatorView.rx.isAnimating).addDi\

3 sposableTo(disposeBag)

We use a similar technique to hide the keyboard. When the user taps the

search or cancel buttons, we resign the search bar as the first responder.

AddLocationViewController.swift

1 searchBar.rx.searchButtonClicked

2 .asDriver(onErrorJustReturn: ())

3 .drive(onNext: { [unowned self] in

4 self.searchBar.resignFirstResponder()

5 })

6 .disposed(by: disposeBag)

7

8 searchBar.rx.cancelButtonClicked

9 .asDriver(onErrorJustReturn: ())

10 .drive(onNext: { [unowned self] in

11 self.searchBar.resignFirstResponder()

12 })

13 .disposed(by: disposeBag)

This means we can remove the implementation of the

UISearchBarDelegate protocol in its entirety. Delegation is a nice pattern,

but it feels great every time I can use Rx to replace boilerplate code like

this.

We could do the same for the UITableViewDataSource and

UITableViewDelegate protocols, but I don’t want to overwhelm you too

much at this point. Build and run Cloudy to make sure we didn’t break

anything during the refactoring operation.

What Have We Accomplished

You may be wondering what we gained by introducing the Model-View-

ViewModel pattern and the AddLocationViewViewModel class in the

AddLocationViewController class. Let’s take a look.

The view controller is no longer in charge of forward geocoding. In fact, it

doesn’t even know about the Core Location framework. That’s our first

accomplishment.

But, more importantly, the view controller no longer manages state. This

is thanks to Rx and the Model-View-ViewModel pattern. The less state

your application manages the better and this is especially true for view

controllers. But what has changed?

User input is funneled to the view model.

The user’s input is directly funneled to the view model. The view model

uses the input of the search bar to perform geocoding requests. The

results of these geocoding requests are streamed back to the view

controller through the locations driver and the view controller’s table

view is updated as a result.

The view model doesn’t keep any state either. In true Rx fashion, it

manages two data streams, a stream of arrays with locations and a

stream of boolean values that indicate whether a geocoding request is in

flight. If you’re new to reactive programming and bindings, then this may

take some getting used to. But I hope you agree that the result is a

welcome improvement.

We also got rid of the UISearchBarDelegate protocol implementation. It’s a

small win but nevertheless welcome.

We’re not quite done yet. In this book, I promised you that testing

becomes easier if you adopt the Model-View-ViewModel pattern. Let’s

put that to the test like we did earlier in this book. But, first, we need to

deal with an obstacle that’s preventing us from writing good unit tests.

24 Protocol Oriented Programming and

Dependency Injection

If we want to test the AddLocationViewViewModel class, we need the ability

to stub the responses of the geocoding requests we make to Apple’s

location services. Only then can we write fast and reliable unit tests.

Being in control of your environment is essential if your goal is creating a

robust test suite.

Not only do we want to be in control of the response we receive from the

geocoding requests, we don’t want the test suite to rely on a service we

don’t control. It can make the test suite slow and unreliable.

But how do we stub the responses of the geocoding requests we make?

The Core Location framework is a system framework. We cannot mock

the CLGeocoder class. The solution is simple, but it requires a bit of work.

A Plan of Action

The solution involves three steps:

First, we need to create a service that’s in charge of performing the

geocoding requests. That service needs to be injected into the view

model. The view model shouldn’t be in charge of instantiating the service.

Second, the service we inject into the view model conforms to a protocol

we define. The protocol is nothing more than a definition of an interface

that allows the view model to initiate a geocoding request. It initiates the

geocoding request, it doesn’t perform the geocoding request.

Third, the service conforms to the protocol and we inject an instance of

the service into the view model.

Not only does this solution decouple the view model from the Core

Location framework, the view model won’t even know which service it’s

using, that is, as long as the service conforms to the protocol we define.

using, that is, as long as the service conforms to the protocol we define.

Don’t worry if this sounds confusing. Let’s start by creating the protocol

for the service. We can draw inspiration from the current implementation

of the AddLocationViewViewModel class. It shows us what the protocol

should look like.

Defining the Protocol

Create a new file in the Protocols group and name it

LocationService.swift.

Creating LocationService.swift

The protocol’s definition will be short. We only define the interface we

need to extract the CLGeocoder class from the view model.

LocationService.swift

1 protocol LocationService {

2

3 }

We first define a type alias, LocationServiceCompletionHandler. This is

primarily for convenience.

LocationService.swift

1 typealias LocationServiceCompletionHandler = ([Location], Error?) ->\

2 Void

More important is the definition of the method that performs the

geocoding request, geocode(addressString:completionHandler:). It

accepts an address string and a completion handler. Because the

geocoding request is performed asynchronously, we mark the completion

handler as escaping. The completion handler is of type

LocationServiceCompletionHandler.

LocationService.swift

1 func geocode(addressString: String?, completionHandler: @escaping Lo\

2 cationServiceCompletionHandler)

Adopting the Protocol

With the LocationService protocol in place, it’s time to create a class that

adopts the LocationService protocol. Create a new group, Services, and

create a class named Geocoder. You can name it whatever you like.

Creating Geocoder.swift

Because we’re going to use the CLGeocoder class to perform the

geocoding requests, we need to import the Core Location framework.

Geocoder.swift

1 import CoreLocation

We define the Geocoder class. The class should conform to the

LocationService protocol we defined earlier.

Geocoder.swift

1 import CoreLocation

2

3 class Geocoder: LocationService {

4

5 }

The Geocoder class has one private, lazy, variable property, geocoder, of

type CLGeocoder.

Geocoder.swift

1 import CoreLocation

2

3 class Geocoder: LocationService {

4

5 // MARK: - Properties

6

7 private lazy var geocoder = CLGeocoder()

8

9 }

The only thing left to do is implement the method of the LocationService

protocol. This isn’t difficult since most of the implementation can be found

in the current implementation of the AddLocationViewViewModel class.

Geocoder.swift

1 // MARK: - Location Service Protocol

2

3 func geocode(addressString: String?, completionHandler: @escaping Lo\

4 cationService.LocationServiceCompletionHandler) {

5 guard let addressString = addressString else {

6 completionHandler([], nil)

7 return

8 }

9

10 // Geocode Address String

11 geocoder.geocodeAddressString(addressString) { (placemarks, erro\

12 r) in

13 if let error = error {

14 completionHandler([], error)

15 print("Unable to Forward Geocode Address (\(error))")

16

17 } else if let \_placemarks = placemarks {

18 // Update Locations

19 let locations = \_placemarks.flatMap({ (placemark) -> Loc\

20 ation? in

21 guard let name = placemark.name else { return nil }

22 guard let location = placemark.location else { retur\

23 n nil }

24 return Location(name: name, latitude: location.coord\

25 inate.latitude, longitude: location.coordinate.longitude)

26 })

27

28 completionHandler(locations, nil)

29 }

30 }

31 }

This should look familiar. The only difference is that we pass the array of

Location instances to the completion handler of the method along with

any errors that pop up.

We now have the ingredients we need to refactor the

AddLocationViewController and AddLocationViewViewModel classes. Let’s

start with the AddLocationViewViewModel class.

Refactoring the Add Location View View Model

Open AddLocationViewViewModel.swift and replace the geocoder

property with a constant property, locationService, of type

LocationService.

AddLocationViewViewModel.swift

1 // MARK: -

2

3 private let locationService: LocationService

This simple change means that the AddLocationViewViewModel class no

longer knows how the application performs the geocoding requests. It

could be the Core Location framework, but it might as well be some other

library. This will come in handy later. It also means we can remove the

import statement for the Core Location framework.

AddLocationViewViewModel.swift

1 import RxSwift

2 import RxCocoa

3 import Foundation

4

5 class AddLocationViewViewModel {

6

7 ...

8

9 }

We inject a location service into the view model using initializer

injection. We pass a second argument to the initializer of the

AddLocationViewViewModel class. The only requirement for the argument

is that it conforms to the LocationService protocol. We set the

locationService property in the initializer.

AddLocationViewViewModel.swift

1 // MARK: - Initialization

2

3 init(query: Driver<String>, locationService: LocationService) {

4 // Set Properties

5 self.locationService = locationService

6

7 query

8 .throttle(0.5)

9 .distinctUntilChanged()

10 .drive(onNext: { [weak self] (addressString) in

11 self?.geocode(addressString: addressString)

12 })

13 .disposed(by: disposeBag)

14 }

We also need to update the geocode(addressString:) method of the

AddLocationViewViewModel class. Because most of the heavy lifting is

done by the location service, the implementation is shorter and simpler.

AddLocationViewViewModel.swift

1 private func geocode(addressString: String?) {

2 guard let addressString = addressString, addressString.character\

3 s.count > 2 else {

4 \_locations.value = []

5 return

6 }

7

8 \_querying.value = true

9

10 // Geocode Address String

11 locationService.geocode(addressString: addressString) { [weak se\

12 lf] (locations, error) in

13 self?.\_querying.value = false

14 self?.\_locations.value = locations

15

16 if let error = error {

17 print("Unable to Forward Geocode Address (\(error))")

18 }

19 }

20 }

Refactoring the Add Location View Controller

The only change we need to make in the AddLocationViewController

class is small. Open AddLocationViewController.swift and navigate to

the viewDidLoad() method. We only need to update the line on which we

initialize the view model.

AddLocationViewViewModel.swift

1 // Initialize View Model

2 viewModel = AddLocationViewViewModel(query: searchBar.rx.text.orEmpt\

3 y.asDriver(), locationService: Geocoder())

That’s it. Build and run the application to make sure we didn’t break

anything. The AddLocationViewViewModel class is now ready to be tested.

25 Testing and Mocking

Setting Up the Environment

It’s time to unit test the AddLocationViewViewModel class. Create a new

unit test case class in the Test Cases group of the CloudyTests target

and name it AddLocationViewViewModelTests.swift.

Creating AddLocationViewViewModelTests.swift

Remove the sample tests and add an import statement for RxSwift,

RxTest, and RxBlocking. As I mentioned earlier, RxTest and

RxBlocking make testing reactive code much easier. It’s one of the best

features of the RxSwift project.

We also need to import the Cloudy module to make sure we have access

to the AddLocationViewViewModel class. Don’t forget to prefix the import

statement with the testable attribute to make internal entities accessible

from within the test target.

AddLocationViewViewModelTests.swift

1 import XCTest

2 import RxTest

3 import RxSwift

4 import RxBlocking

5 @testable import Cloudy

6

7 class AddLocationViewViewModelTests: XCTestCase {

8

9 // MARK: - Set Up & Tear Down

10

11 override func setUp() {

12 super.setUp()

13 }

14

15 override func tearDown() {

16 super.tearDown()

17 }

18

19 }

Click the diamond in the gutter on the left to run the unit tests for the

AddLocationViewViewModel class. We don’t have any tests yet, but it

ensures everything is set up correctly.

Before we can write any tests, we need to declare a few properties:

viewModel of type AddLocationViewViewModel!

scheduler of type SchedulerType!

query of type Variable<String>!

AddLocationViewViewModelTests.swift

1 // MARK: - Properties

2

3 var viewModel: AddLocationViewViewModel!

4

5 // MARK: -

6

7 var scheduler: SchedulerType!

8

9 // MARK: -

10

11 var query: Variable<String>!

Notice that these properties are implicitly unwrapped optionals.

Remember that safety isn’t a major concern when we write unit tests. If

something goes wrong, it means we made a mistake in the test case,

which we need to fix first.

The viewModel property is the view model we’ll be testing. The scheduler

property is less important for our discussion. Schedulers provide a layer

of abstraction for scheduling operations using RxSwift. That’s all you

need to know about schedulers to follow along.

We use the query property to mock user input. Because the view model

doesn’t know where the input comes from, it’s easy to mock user input by

emitting events with the help of the query variable.

We set everything up in the setUp() method of the test case. Remember

that this method is executed every time a unit test is run. It ensures we

start with a clean slate before a unit test is run.

In the setUp() method, we create a Variable of type String and assign it

to the query property. The variable is initialized with an empty string.

AddLocationViewViewModelTests.swift

1 override func setUp() {

2 super.setUp()

3

4 // Initialize Query

5 query = Variable<String>("")

6 }

Mocking the Location Service

The next step is initializing the view model. But we have a problem. If we

use the Geocoder class we created earlier, we cannot stub the response

of the geocoding request. Remember that we want to control the

environment in which the test suite is run. If the application talks to a

location service, we need the ability to control its response.

Fortunately, we already did the heavy lifting to make this very easy. All

we need to do is create a mock location service. We start by declaring a

private, nested class, MockLocationService, which conforms to the

LocationService protocol.

AddLocationViewViewModelTests.swift

1 import XCTest

2 import RxTest

3 import RxSwift

4 import RxBlocking

5 @testable import Cloudy

6

7 class AddLocationViewViewModelTests: XCTestCase {

8

9 private class MockLocationService: LocationService {

10

11 }

12

13 ...

14

15 }

The only method we need to implement to conform to the

LocationService protocol is geocode(addressString:completionHandler:).

If addressString has a value and its value isn’t an empty string, we invoke

the completion handler with an array containing one Location instance.

The second argument of the completion handler, an optional error, is nil.

Notice that we control what the location service returns. In this example,

we return a Location instance with a name of Brussels and a fixed set of

coordinates.

AddLocationViewViewModelTests.swift

1 private class MockLocationService: LocationService {

2

3 func geocode(addressString: String?, completionHandler: @escapin\

4 g LocationServiceCompletionHandler) {

5 if let addressString = addressString, !addressString.isEmpty\

6 {

7 // Create Location

8 let location = Location(name: "Brussels", latitude: 50.8\

9 503, longitude: 4.3517)

10

11 // Invoke Completion Handler

12 completionHandler([location], nil)

13 } else {

14 // Invoke Completion Handler

15 completionHandler([], nil)

16 }

17 }

18

19 }

If addressString has no value or its value is equal to an empty string, we

invoke the completion handler with an empty array. The second

argument of the completion handler, an optional error, is nil.

That’s it. In the setUp() method, we can now instantiate an instance of

the MockLocationService class and pass it as an argument to the

initializer of the AddLocationViewViewModel class. The first argument of the

initializer is the query property as a driver.

AddLocationViewViewModelTests.swift

1 override func setUp() {

2 super.setUp()

3

4 // Initialize Query

5 query = Variable<String>("")

6

7 // Initialize Location Service

8 let locationService = MockLocationService()

9

10 // Initialize View Model

11 viewModel = AddLocationViewViewModel(query: query.asDriver(), lo\

12 cationService: locationService)

13 }

We also create a concurrent dispatch queue scheduler and assign it to

the scheduler property. Don’t worry about this if you’re not familiar with

RxSwift.

AddLocationViewViewModelTests.swift

1 override func setUp() {

2 super.setUp()

3

4 // Initialize Query

5 query = Variable<String>("")

6

7 // Initialize Location Service

8 let locationService = MockLocationService()

9

10 // Initialize View Model

11 viewModel = AddLocationViewViewModel(query: query.asDriver(), lo\

12 cationService: locationService)

13

14 // Initialize Scheduler

15 scheduler = ConcurrentDispatchQueueScheduler(qos: .default)

16 }

Writing Unit Tests

The first test we’re going to write tests the locations driver of the view

model. Now that we control what the location service returns, we can test

the behavior of the view model. We name the test

testLocations\_HasLocations().

AddLocationViewViewModelTests.swift

1 // MARK: - Tests for Locations

2

3 func testLocations\_HasLocations() {

4

5 }

We create an observable and store a reference to it in a constant named

observable.

AddLocationViewViewModelTests.swift

1 // Create Subscription

2 let observable = viewModel.locations.asObservable().subscribeOn(sche\

3 duler)

We emit a new event by setting the value of the query property. This

mimics the user entering a name of a city in the search bar of the add

location view controller.

AddLocationViewViewModelTests.swift

1 // Set Query

2 query.value = "Brus"

We use toBlocking() to make the test synchronous. The result is stored

in the result constant.

AddLocationViewViewModelTests.swift

1 // Fetch Result

2 let result = try! observable.skip(1).toBlocking().first()!

The result is an array of Location instances. We assert that result isn’t

equal to nil and that it contains one element.

AddLocationViewViewModelTests.swift

1 XCTAssertNotNil(result)

2 XCTAssertEqual(result.count, 1)

We also include an assertion for the location stored in the array by

making sure the name property of the location is equal to Brussels.

AddLocationViewViewModelTests.swift

1 // Fetch Location

2 let location = result.first!

3

4 XCTAssertEqual(location.name, "Brussels")

Notice that we make ample use of the exclamation mark. If anything

Notice that we make ample use of the exclamation mark. If anything

blows up, it’s because of a failed test or an error we made. In other

words, we made a mistake we need to fix. This is what the unit test looks

like.

AddLocationViewViewModelTests.swift

1 // MARK: - Tests for Locations

2

3 func testLocations\_HasLocations() {

4 // Create Subscription

5 let observable = viewModel.locations.asObservable().subscribeOn(\

6 scheduler)

7

8 // Set Query

9 query.value = "Brus"

10

11 // Fetch Result

12 let result = try! observable.skip(1).toBlocking().first()!

13

14 XCTAssertNotNil(result)

15 XCTAssertEqual(result.count, 1)

16

17 // Fetch Location

18 let location = result.first!

19

20 XCTAssertEqual(location.name, "Brussels")

21 }

We also need to unit test the behavior when the user enter’s an empty

string. This is very similar. We name this test

testLocations\_NoLocations(). I won’t go into the finer details of RxSwift,

such as drivers replaying the last event when a subscriber is added. We

expect an empty array of locations and that’s what we test.

AddLocationViewViewModelTests.swift

1 func testLocations\_NoLocations() {

2 // Create Subscription

3 let observable = viewModel.locations.asObservable().subscribeOn(\

4 scheduler)

5

6 // Fetch Result

7 let result: [Location] = try! observable.toBlocking().first()!

8

9 XCTAssertNotNil(result)

10 XCTAssertEqual(result.count, 0)

11 }

Let’s run the unit tests we have so far to make sure they pass. That’s

looking good.

Running the Unit Tests

While I won’t be discussing every unit test of the

AddLocationViewViewModel class, I want to show you a few more. With the

next unit test, we test the location(at:) method of the

AddLocationViewViewModel class.

The test looks similar to the ones we wrote earlier. Instead of inspecting

the value of locations, we ask the view model for the location at index 0.

It shouldn’t be equal to nil. We also assert that the name of the Location

instance is equal to Brussels.

AddLocationViewViewModelTests.swift

1 // MARK: - Tests for Location At Index

2

3 func testLocationAtIndex\_NonNil() {

4 // Create Subscription

5 let observable = viewModel.locations.asObservable().subscribeOn(\

6 scheduler)

7

8 // Set Query

9 query.value = "Brus"

10

11 // Fetch Result

12 let \_ = try! observable.skip(1).toBlocking().first()!

13

14 // Fetch Location

15 let result = viewModel.location(at: 0)

16

17 XCTAssertNotNil(result)

18 XCTAssertEqual(result!.name, "Brussels")

19 }

We can create a similar test for an index that is out of bounds. In that

case, the location(at:) method returns nil, which we can test for.

AddLocationViewViewModelTests.swift

1 func testLocationAtIndex\_Nil() {

2 // Create Subscription

3 let observable = viewModel.locations.asObservable().subscribeOn(\

4 scheduler)

5

6 // Set Query

7 query.value = "Brus"

8

9 // Fetch Result

10 let \_ = try! observable.skip(1).toBlocking().first()!

11

12 // Fetch Location

13 let result = viewModel.location(at: 1)

14

15 XCTAssertNil(result)

16 }

You can find the other unit tests for the AddLocationViewViewModel class

in the completed project of this chapter. I hope it’s clear that writing unit

tests for a view model with bindings isn’t that much more complicated.

26 Where to Go From Here

You’ve reached the end of the book and you should now have a good

understanding of the Model-View-ViewModel pattern and how it

compares to the Model-View-Controller pattern. In this book, we

refactored Cloudy. We transitioned the project from the Model-View-

Controller pattern to the Model-View-ViewModel pattern. But what did we

gain? Was it worth the effort?

Putting the View Controllers On a Diet

First and foremost, the view controllers of the project have become

skinnier, lightweight, and focused. They no longer deal with data

manipulation. In fact, the view controllers are not aware of or keep a

reference to the models used in the project.

Keep in mind that the goal of the Model-View-ViewModel pattern isn’t

merely removing code from view controllers and dumping it into a view

model. The goal is more ambitious and some benefits are more subtle.

The week view controller doesn’t keep a reference to the weather data. It

now uses a view model instead. It doesn’t declare properties for date

formatters to format dates. They’re no longer needed. And, the most

substantial change, it’s no longer responsible for configuring the table

view cells of the table view. The view model hands the table view a view

model for each table view cell and, with the help of a protocol, the latter

knows exactly how to configure itself using the view model.

You also learned that the Model-View-ViewModel pattern can be used in

view controllers that aren’t driven by data. The settings view controller is

an example of this. It doesn’t use a model to populate itself. It merely

shows the user a table view with their preferences. The Model-View-

ViewModel pattern is a good fit for almost any type of view controller.

The add location view controller also underwent a dramatic change.

Before its facelift, it was in charge of handling user interaction, data

Before its facelift, it was in charge of handling user interaction, data

manipulation, forward geocoding, and data visualization. That’s no longer

true after the refactoring operation. We gained a number of key benefits.

The view model is in charge of performing forward geocoding requests

and the view controller no longer manages state. Nor does the view

model. Thanks to reactive programming, we only deal with streams of

data. That’s a significant change.

The add location view controller is focused and lightweight. It’s only task

is handling user interaction and displaying data to the user. In other

words, it’s a view controller in the purest sense. That’s something we

accomplished by implementing the Model-View-ViewModel pattern.

Introducing View Models

The code we removed from the view controllers now lives in the view

models we introduced in the project. What I like about these view models

is their simplicity. In their most basic form, the view models convert the

raw values of the model they manage to values the view controller can

directly display in the view it manages. Even though the add location view

view model is a bit more advanced, it remains focused.

We also included the logic of the user’s preferences in several view

models. The result is that the interface of the view models is clear and

concise. Using the view models in the view controllers and the table view

cells is as simple as asking for a value to display. That’s the essence of

the Model-View-ViewModel pattern.

Improved Testability

I have to admit that I don’t like writing unit tests, but this isn’t much of a

problem when testing view models. The reason is simple, writing unit

tests for view models is easy. Because you can carefully control the

model that’s used to instantiate the view model, unit testing becomes

almost painless.

And remember that, if we were to write unit tests for the view controllers

to unit test the same functionality, we would have to deal with a bunch of

issues and the setup would look more complex. Improved testability is

another key benefit of the Model-View-ViewModel pattern. I hope this

another key benefit of the Model-View-ViewModel pattern. I hope this

book has convinced you of this.

More Flexibility

It’s no coincidence that the pattern is named Model-View-ViewModel and

not Mode-View-Controller-ViewModel. The controller still plays a part, but

you need to understand that the Model-View-ViewModel pattern is quite

flexible. The implementation of the week view controller shows this.

The week view controller asks its view model for a view model for a

weather day table view cell. The view controller isn’t involved in

configuring the table view cell itself. The Model-View-ViewModel pattern

helps configure views with view models. The view controller is still

involved, but its role can sometimes be minimal.

Just remember what the Model-View-ViewModel pattern tries to

accomplish and don’t try to think in terms of the Model-View-Controller

pattern too much.

Where to Start

You now have the knowledge to create applications that use the Model-

View-ViewModel pattern. But chances are that you have one or more

projects that use the Model-View-Controller pattern. This isn’t a problem,

though. It’s easy to start with the Model-View-ViewModel pattern in an

existing project. You don’t need to spend weeks or months refactoring.

Start small. For existing projects, you don’t need to choose between MVC

and MVVM. Remember that MVVM is in many ways similar to MVC.

I hope you enjoyed this book on the Model-View-ViewModel pattern. If

you have any questions or feedback, reach out to me via email

(bart@cocoacasts.com) or Twitter (@\_bartjacobs). I’m here to help.\_