Mastering MVVM With Swift

Written by Bart Jacobs

Cocoacasts

Mastering MVVM With Swift

Bart Jacobs

This book is for sale at http://leanpub.com/mastering-mvvm-with-swift

This version was published on 2017-07-18

*

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

*

Table of Contents

Welcome

Welcome to **Mastering MVVM With Swift**. I'm glad to see you here. In this book, you learn the ins and outs of the **Model-View-ViewModel** pattern. The goal of this book is to provide you with the ingredients you need to implement the Model-View-ViewModel pattern in your own projects.

Xcode 9 and Swift 4

This book uses Xcode 9 and Swift 4. Only the chapters about reactive programming stick with Xcode 9 and Swift 3. These chapters will be updated once RxSwift/RxCocoa officially support Swift 4. If you want to follow along, make sure you have Xcode 8 or 9 installed on your machine. Everything you learn in this book applies to both Swift 3 and Swift 4.

What You'll Learn

This book covers much more than the **Model-View-ViewModel** pattern. We start with an overview of the Model-View-ViewModel pattern and we compare it with the popular **Model-View-Controller** pattern, a pattern you're probably already familiar with.

In the remainder of the book, we refactor **Cloudy**, a weather application powered by the Model-View-Controller pattern. We refactor Cloudy to use the Model-View-ViewModel pattern instead. This will show you how to apply the Model-View-ViewModel pattern in a production application. The refactoring operation will show you exactly what needs to change to move from MVC to MVVM, highlighting the benefits and challenges that go with this migration.

Along the way, you learn what view models are, how to create them, and how to use them in view controllers. We further simplify the view controllers of the project using protocol-oriented programming. Protocols and MVVM work very well together.

and MVVM work very well together.

Later in the book, we write unit tests for the view models we created. One of the key benefits of the Model-View-ViewModel pattern is improved testability and that's something I want to show you first-hand. Writing unit tests for view models is really easy.

The Model-View-ViewModel pattern really shines with the help of bindings. I first show you how to create a custom bindings solution. This is an important step as it will show you how the Model-View-ViewModel pattern and bindings work under the hood.

Later in the book, we take it a step further by taking advantage of RxSwift and RxCocoa. You don't need to be familiar with reactive programming to understand these chapters. We primarily focus on the Model-View-ViewModel pattern and how it plays together with bindings. The Model-View-ViewModel pattern works with any bindings solution.

We end this book with a quick recap of what we gained from using the Model-View-ViewModel pattern instead of the Model-View-Controller pattern. The changes we apply to Cloudy are pretty dramatic and I'm sure you'll appreciate the benefits the Model-View-ViewModel pattern has to offer.

This book covers a lot of ground, but I'm here to guide you along the way. If you have any feedback or questions, reach out to me via email (bart@cocoacasts.com) or Twitter (@_bartjacobs). I'm here to help.

How to Use This Book

If you'd like to follow along, I recommend downloading the source files that come with this book. The chapters that include code each have a starter project and a finished project. This makes it easy to follow along or pick a random chapter from the book. Click here to download the source files for this book. If you're new to the Model-View-ViewModel pattern, then I recommend reading every chapter of the book.

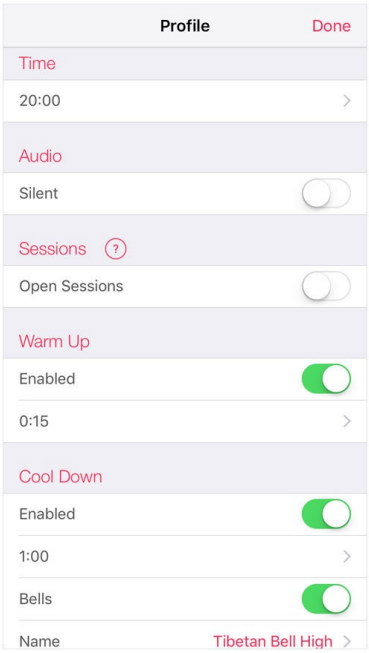
Not everyone likes books. If you prefer video, then you may be interested in a video course in which I teach the Model-View-ViewModel pattern.

The content is virtually identical. The only difference is that you can see how I refactor Notes using the Model-View-ViewModel pattern. You can find the video course on the Cocoacasts website.

1 Is MVC Dead

Model-View-Controller, or **MVC** for short, is a widely used design pattern for architecting software applications. Cocoa applications are centered around the Model-View-Controller pattern and many of Apple's frameworks make heavy use of the Model-View-Controller pattern.

Last year, I was working on the next major release of Samsara, a meditation application I've been developing for the past few years. The settings view is an important aspect of the application.



Samsara's Settings View

From the perspective of the user, the settings view is nothing more than a collection of controls, labels, and buttons. Under the hood, however, lives a fat view controller responsible for managing the content of the table view and the data that's fed to the table view.

Table views are flexible and cleverly designed. A table view asks its data source for the data it needs to present and it delegates user interaction to its delegate. That makes them incredibly reusable. Unfortunately, the

more the table view gains in complexity, the more unwieldy the data source becomes.

Table views are a fine example of the Model-View-Controller pattern in action. The model layer hands the data source (mostly a view controller) the data the view layer (the table view) needs to display. But table views also illustrate how the Model-View-Controller pattern can, and very often does, fall short. Before we take a closer look at the problem, I'd like to take a brief look at the Model-View-Controller pattern. What is it, what makes it so popular, and, more importantly, what are its drawbacks?

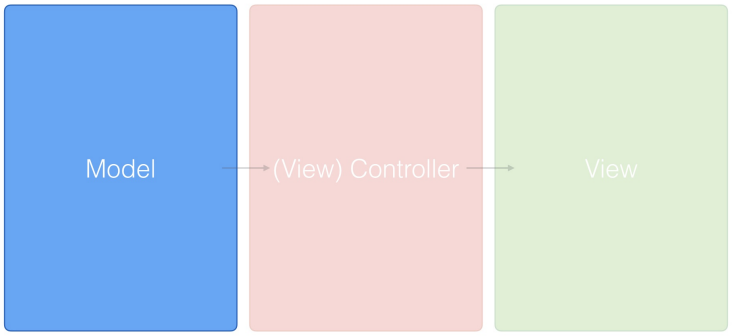
What Is It?

The MVC pattern breaks an application up into three components or layers:

- **M**odel
- **V**iew
- **C**ontroller

Model

The model layer is responsible for the business logic of the application. It manages the application state. This also includes reading and writing data, persisting application state, and it may even include tasks related to data management, such as networking and data validation.



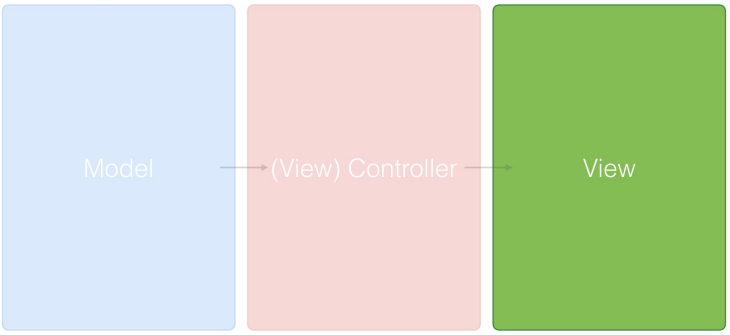
The M In MVC

View

The view layer has two important tasks:

- presenting data to the user
- handling user interaction

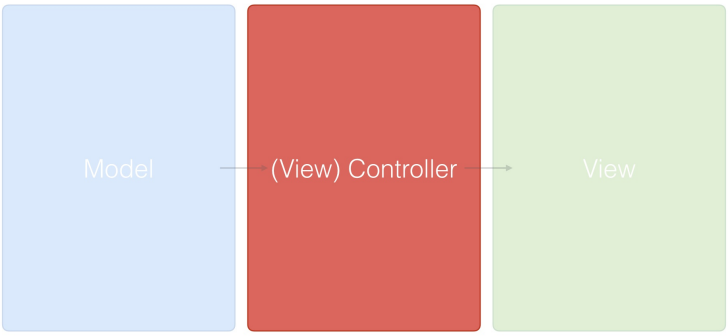
A core principle of the MVC pattern is the view layer's ignorance with respect to the model layer. Views are dumb objects. They only know how to present data to the user. They don't know or understand *what* they're presenting. This makes them flexible and easy to reuse.



The V In MVC

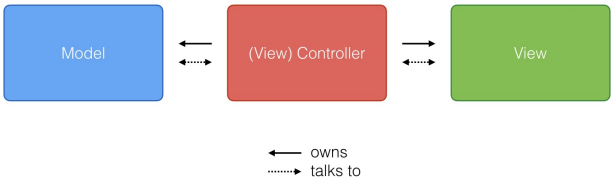
Controller

The view layer and the model layer are glued together by one or more controllers. In an iOS application, that glue is a view controller, an instance of the `UIViewController` class or a subclass thereof. In a macOS application, that glue is a window controller, an instance of the `NSWindowController` class or a subclass thereof.



The C In MVC

A controller knows about the view layer as well as the model layer. This often results in tight coupling, making controllers the least reusable components of an application based on the Model-View-Controller pattern. The view and model layers don't know about the controller. The controller owns the views and the models it interacts with.



Model-View-Controller in a Nutshell

Advantages

Separation of Concerns

The advantage of the MVC pattern is a clear separation of concerns. Each layer of the Model-View-Controller pattern is responsible for a clearly defined aspect of the application. In most applications, there's no confusion about what belongs in the view layer and what belongs in the model layer.

What goes into controllers is often less clear. The result is that controllers

What goes into controllers is often less clear. The result is that controllers are frequently used for everything that doesn't clearly belong in the view layer or the model layer.

Reusability

While controllers are often not reusable, view and model objects are mostly easy to reuse. If the Model-View-Controller pattern is correctly implemented, the view layer and the model layer should be composed of reusable components.

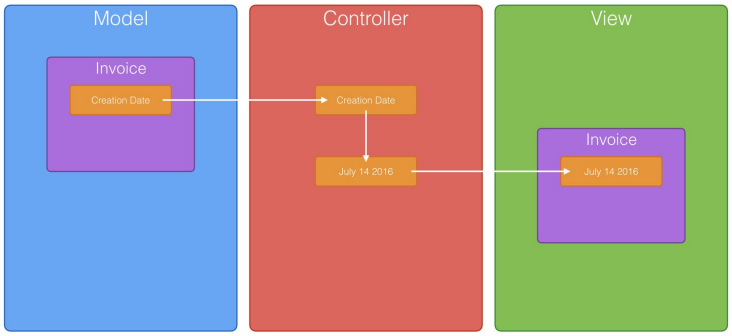
Problems

If you've spent any amount of time reading books or tutorials about iOS or macOS development, then you've probably come across people complaining about the Model-View-Controller pattern. Why is that? What's wrong with the Model-View-Controller pattern?

A clear separation of concerns is great. It makes your life as a developer easier. Projects are easier to architect and structure. But that's only part of the story. A lot of the code you write doesn't belong in the view layer or the model layer. No problem. Dump it in the controller. Problem solved. Right? Not really.

An Example

Data formatting is a common task. Imagine that you're developing an invoicing application. Each invoice has a creation date. Depending on the locale of the user, the date of an invoice needs to be formatted differently.



An Example

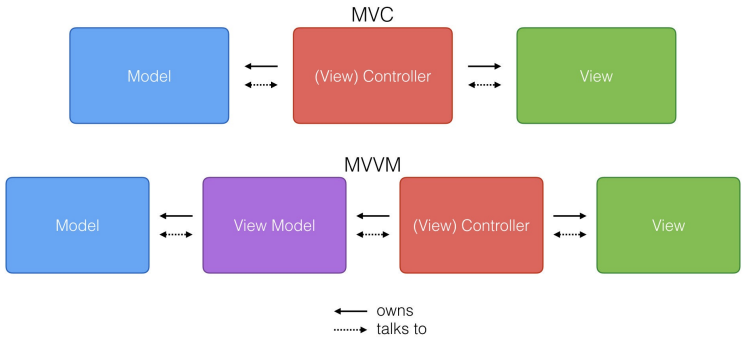
The creation date of an invoice is stored in the model layer and the view displays the formatted date. That's obvious. But who's responsible for formatting the date? The model? Maybe. The view? Remember that the view shouldn't need to understand what it's presenting to the user. But why should the model be responsible for a task related to the user interface?

Wait a minute. What about our good old controller? Sure. Dump it in the controller. After thousands of lines of code, you end up with a bunch of overweight controllers, ready to burst and impossible to test. Isn't MVC the best thing ever?

How Can We Solve This?

In recent years, another pattern has been gaining traction in the Cocoa community. It's commonly referred to as the Model-View-ViewModel pattern, MVVM for short. The origins of the MVVM pattern lead back to Microsoft's .NET framework and it continues to be used in modern Windows development.

How does the Model-View-ViewModel pattern solve the problem we described earlier? The Model-View-ViewModel pattern introduces a fourth component, the **view model**. The view model is responsible for managing the model and funneling the model's data to the view via the controller. This is what that looks like.



Model-View-ViewModel in a Nutshell

Despite its name, the MVVM pattern includes four components or layers:

- **M**odel
- **V**iew
- **V**iew **M**odel
- Controller

The implementation of a view model is often straightforward. All it does is translate data from the model to values the view layer can display. The controller is no longer responsible for this ungrateful task. Because view models have a close relationship with the models they consume, they're often considered more model than view.

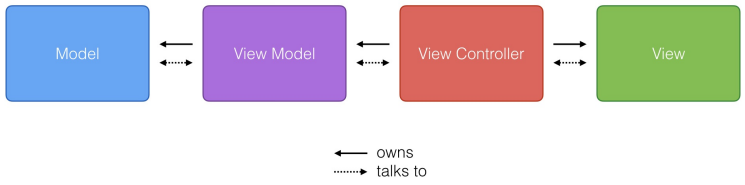
In the next chapter, we take a closer look at the internals of the Model-View-ViewModel pattern.

2 How Does MVVM Work

In this chapter, we take a closer look at the internals of the Model-View-ViewModel pattern. We explore what MVVM is and how it works.

Remember form the previous chapter that the Model-View-ViewModel pattern consists of four components or layers:

- **M**odel
- **V**iew
- **V**iew **M**odel
- Controller



The Model-View-ViewModel Pattern in A Nutshell

Keep this diagram in mind. Let's start by taking a look at the advantages MVVM has over MVC. Why would you even consider trading the Model-View-Controller pattern for the Model-View-ViewModel pattern?

Advantages of MVVM

We already know that the Model-View-Controller pattern has a few flaws. With that in mind, what are the advantages MVVM has over MVC?

Better Separation of Concerns

Let me start by asking you a simple question. What do you do with code

that doesn't fit or belong in the view or model layer? Do you put it in the controller layer? Don't feel guilty, though. This is what most developers do. The problem is that it inevitably leads to fat controllers that are difficult to test and manage.

The Model-View-ViewModel pattern presents a better separation of concerns by adding view models to the mix. The view model translates the data of the model layer into something the view layer can use. The controller layer is no longer responsible for this task.

Improved Testability

View (iOS) and window (macOS) controllers are notoriously hard to test because of their close relation to the view layer. By migrating some responsibilities, such as data manipulation, to the view model, testing becomes much easier. As you'll learn in this book, testing view models is surprisingly easy. Testing? Easy? Absolutely.

Because a view model doesn't have a reference to the view controller that owns it, it's easy to write unit tests for a view model. Another benefit of MVVM is improved testability of view and window controllers. The controller no longer depends on the model layer, which makes them easier to test.

Transparent Communication

The responsibilities of the controller are reduced to controlling the interaction between the view and model layer. The view model provides a transparent interface to the view controller, which it uses to populate the view layer and interact with the model layer. This results in a transparent communication between the four components or layers of your application.

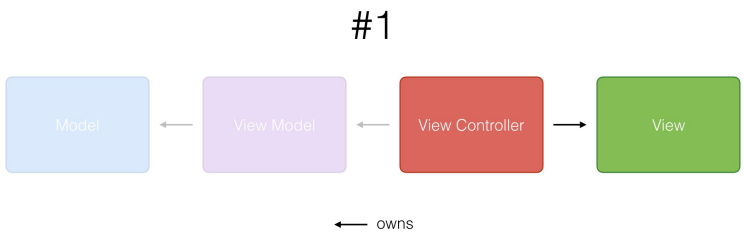
Basic Rules

Before we start implementing the Model-View-ViewModel pattern in an application, I'd like to highlight six key elements that define the Model-View-ViewModel pattern. I sometimes refer to these as *rules*. But once

you understand how the Model-View-ViewModel pattern does its magic, it’s fine to bend or break some of these rules.

Rule #1

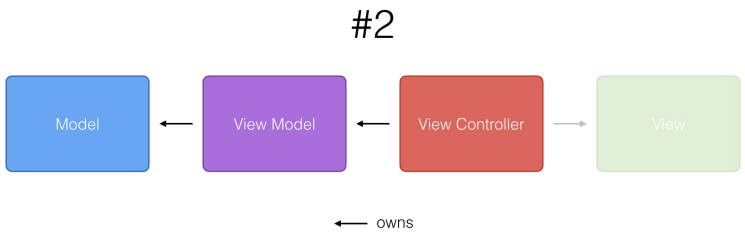
First, the view doesn’t know about the view controller it’s owned by. Remember that views are supposed to be dumb. They only know how to present what they’re given by the view controller to the user. This is a rule you should never break. Ever.



The view doesn’t know about the view controller it’s owned by.

Rule #2

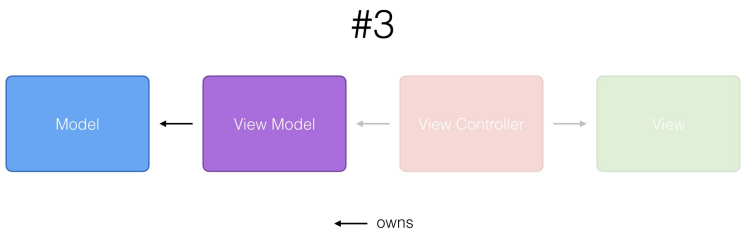
Second, the view or window controller doesn’t know about the model. This is something that separates MVC from MVVM.



The view controller doesn’t know about the model.

Rule #3

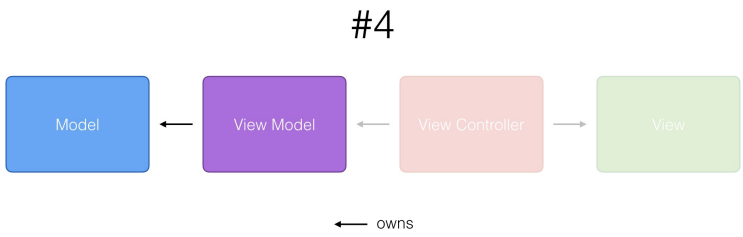
The model doesn’t know about the view model it’s owned by. This is another rule that should never be broken. The model should have no clue who it’s owned by.



The model doesn’t know about the view model it’s owned by.

Rule #4

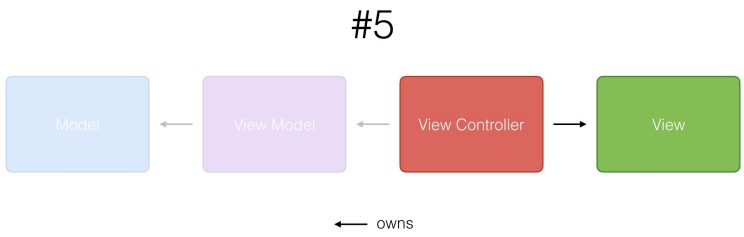
The view model owns the model. In a Model-View-Controller application, the model is usually owned by the view or window controller.



The view model owns the model.

Rule #5

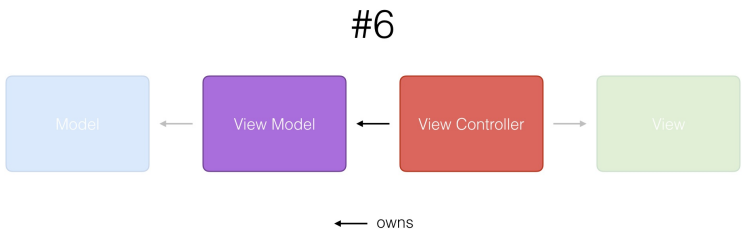
The view or window controller owns the view or window. This relationship remains unchanged.



The view controller owns the view.

Rule #6

And finally, the controller owns the view model. It interacts with the model layer through one or more view models.



The controller owns the view model.

It’s Time to Refactor

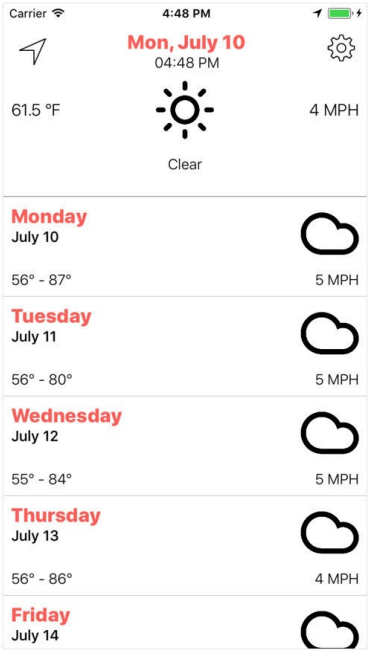
You now know enough about the Model-View-ViewModel pattern to use it in an application. In the remainder of this book, we refactor an existing application. The application is powered by the Model-View-Controller pattern and we refactor it in such as way that it uses the Model-View-ViewModel pattern instead. Let’s get started.

3 Meet Cloudy

In the remainder of this book, we're going to refactor an application that's built with MVC and make it adopt MVVM instead. This will teach you two important lessons:

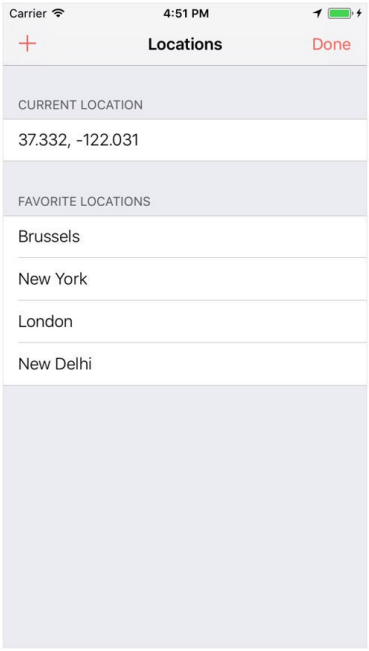
- What are the shortcomings of MVC?
- How can MVVM help resolve these shortcomings?

The application we're going to refactor is **Cloudy**. Cloudy is a lightweight weather application that shows the user the weather of their current location or a saved location. It shows the current weather conditions and a forecast for the next few days. The weather data is retrieved from the Dark Sky API, an easy-to-use weather service.

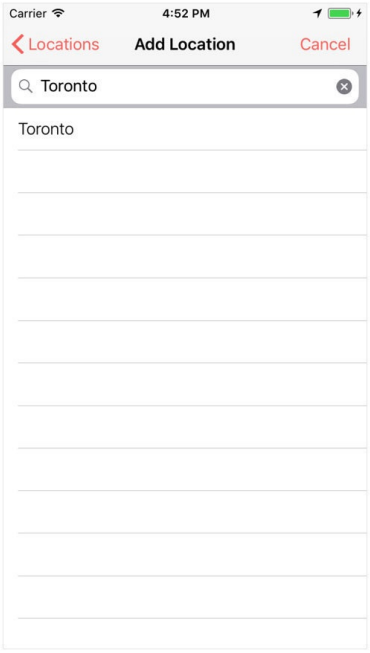


Meet Cloudy

The user can add locations and switch between locations by bringing up the locations view controller.

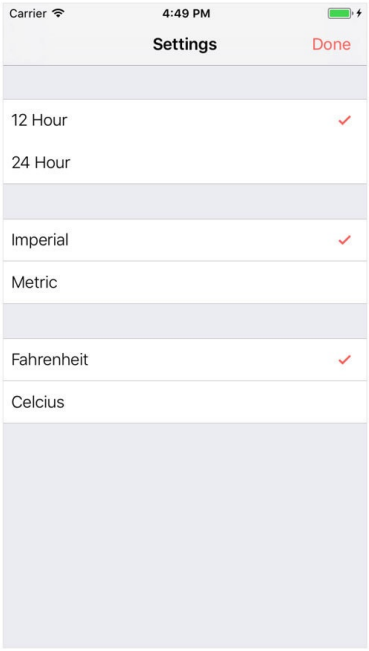


Managing Locations



Adding Locations

Cloudy has a settings view to change the time notation, the application's units system, and the user can switch between degrees Fahrenheit and degrees Celcius.



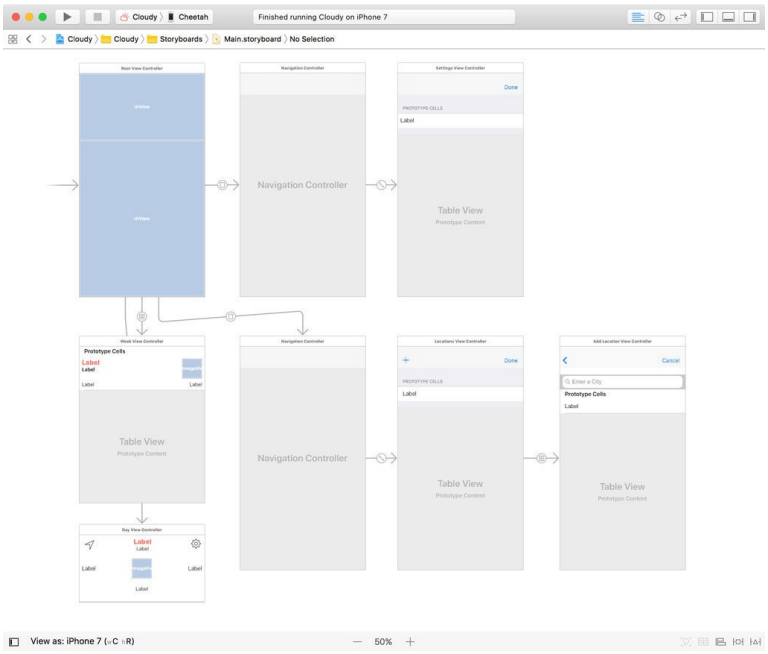
Managing the Application's Settings

Application Architecture

In this chapter, I walk you through the source code of Cloudy. You can follow along by opening the project of this chapter.

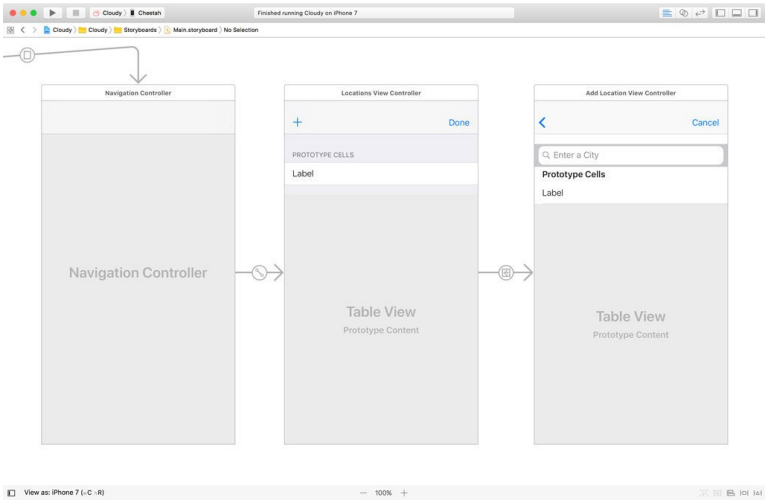
Storyboard

The main storyboard is the best place to start. You can see that we have a container view controller with two child view controllers. The top child view controller shows the current weather conditions, the bottom child view controller displays the forecast for the next few days in a table view.



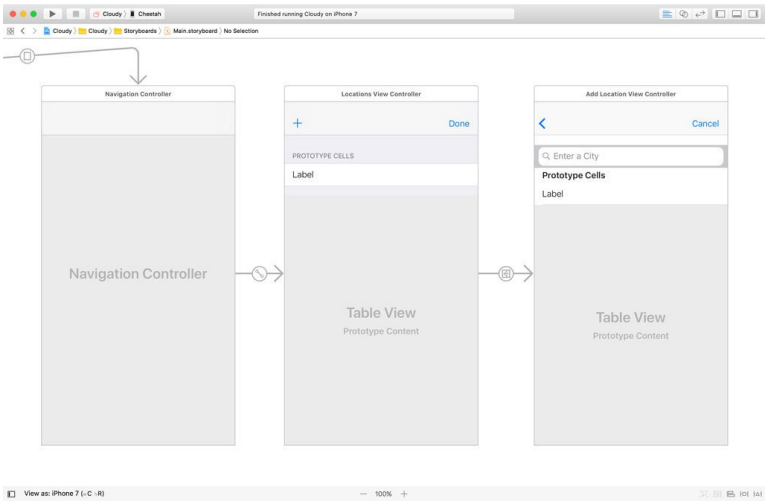
Cloudy's Main Storyboard

If the user taps the location button in the top child view controller (top left), the locations view controller is shown. The user can switch between locations and add new locations using the add location view controller.



Cloudy's Main Storyboard

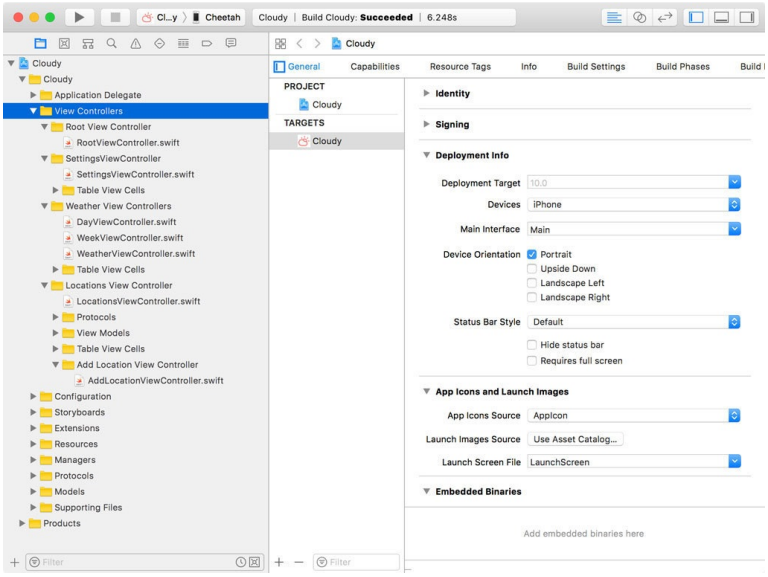
If the user taps the settings button in the top child view controller (top right), the settings view controller is shown. This is another table view listing the options we discussed earlier.



Cloudy's Main Storyboard

View Controllers

If we open the **View Controllers** group in the **Project Navigator**, we can see the view controller classes that correspond with what I just showed you in the storyboard.



Cloudy's View Controllers

The `RootViewController` class is the container view controller. The `DayViewController` is the top cild view controller and the `WeekViewController` is the bottom child view controller. The `WeatherViewController` class is the superclass of the `DayViewController` and the `WeekViewController`.

Root View Controller

The root view controller is responsible for several tasks:

- it fetches the weather data
- it fetches the current location of the user's device
- it sends the weather data to its child view controllers

The root view controller delegates the fetching of the weather data to the `DataManager` class. This class sends the request to the Dark Sky API and converts the JSON response to model objects. I use a simple, lightweight JSON parser for this task. The implementation of the JSON parser and the `DataManager` class are unimportant for this discussion.

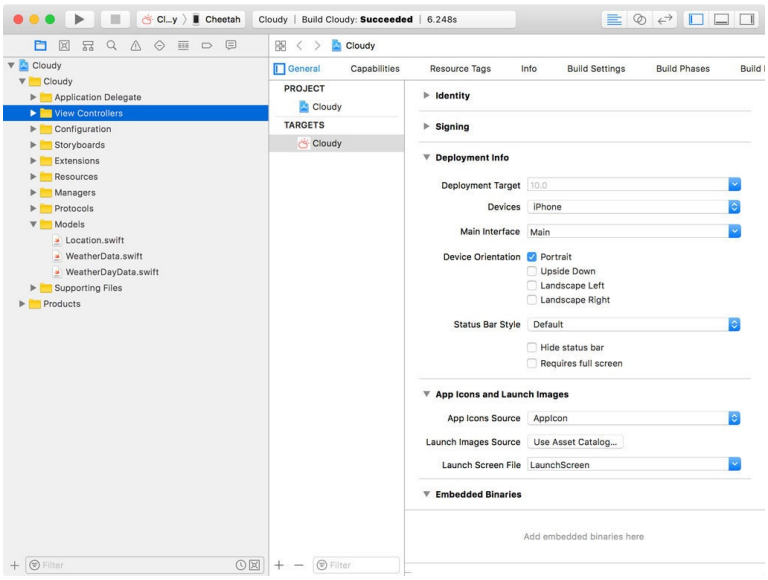
In the completion handler of the `weatherDataForLocation(latitude:longitude:completion:)` method of the `RootViewController` class, the weather data is sent to the day view controller and the week view controller.

RootViewController.swift

```
dataManager.weatherDataForLocation(latitude: latitude, longitude: lo\
ngitude) { (response, error) in
    if let error = error {
        print(error)
    } else if let response = response {
        // Configure Day View Controller
        self.dayViewController.now = response

        // Configure Week View Controller
        self.weekViewController.week = response.dailyData
    }
}
```

Model Objects

The model objects we'll be working with are `Location`, `WeatherData` and `WeatherDayData`. You can find them in the **Models** group.



Model Objects

The `Location` structure makes working with locations a bit easier. There's no magic involved. The `WeatherData` and `WeatherDayData` structures contain the weather data that's fetched from the Dark Sky API. Notice that a `WeatherData` object contains an array of `WeatherDayData` instances.

WeatherData.swift

```
1  import Foundation
2
3  struct WeatherData {
4
5      let time: Date
6
7      let lat: Double
8      let long: Double
9      let windSpeed: Double
10     let temperature: Double
11
12     let icon: String
13     let summary: String
14
15     let dailyData: [WeatherDayData]
16
17 }
```

The current weather conditions are stored in the `WeatherData` object and the forecast for the next few days is stored in an array of `WeatherDayData` objects.

The root view controller only hands the week view controller the array of `WeatherDayData` objects, which it displays in a table view.

WeekViewController.swift

```
var week: [WeatherDayData]?
```

The day view controller receives the `WeatherData` object from the root view controller.

DayViewController.swift

```
var now: WeatherData?
```

Day View Controller

The `now` property of the `DayViewController` class stores the `WeatherData` object. Every time this property is set, the user interface is updated with new weather data by invoking `updateView()`.

DayViewController.swift

```
var now: WeatherData? {
    didSet {
        updateView()
    }
}
```

In `updateView()`, we hide the activity indicator view and update the weather data container, this is nothing more than a view that contains the views displaying the weather data.

DayViewController.swift

```
private func updateView() {
    activityIndicatorView.stopAnimating()

    if let now = now {
        updateWeatherDataContainer(withWeatherData: now)

    } else {
        messageLabel.isHidden = false
        messageLabel.text = "Cloudy was unable to fetch weather data\
."

    }
}
```

The implementation of `updateWeatherDataContainer(withWeatherData:)` is a classic example of the Model-View-Controller pattern. The model object

is torn apart and the raw values are transformed and formatted for display to the user.

DayViewController.swift

```
private func updateWeatherDataContainer(withWeatherData weatherData:\
 WeatherData) {
    weatherDataContainer.isHidden = false

    var windSpeed = weatherData.windSpeed
    var temperature = weatherData.temperature

    let dateFormatter = DateFormatter()
    dateFormatter.dateFormat = "EEE, MMMM d"
    dateLabel.text = dateFormatter.string(from: weatherData.time)

    let timeFormatter = DateFormatter()

    if UserDefaults.timeNotation() == .twelveHour {
        timeFormatter.dateFormat = "hh:mm a"
    } else {
        timeFormatter.dateFormat = "HH:mm"
    }

    timeLabel.text = timeFormatter.string(from: weatherData.time)

    descriptionLabel.text = weatherData.summary

    if UserDefaults.temperatureNotation() != .fahrenheit {
        temperature = temperature.toCelcius()
        temperatureLabel.text = String(format: "%.1f °C", temperatur\
e)
    } else {
        temperatureLabel.text = String(format: "%.1f °F", temperatur\
e)
    }

    if UserDefaults.unitsNotation() != .imperial {
        windSpeed = windSpeed.toKPH()
        windSpeedLabel.text = String(format: "%.f KPH", windSpeed)
    } else {
        windSpeedLabel.text = String(format: "%.f MPH", windSpeed)
    }

    iconImageView.image = imageForIcon(withName: weatherData.icon)
}
```

Week View Controller

The week view controller looks similar in several ways. The `week` property stores the weather data and every time the property is set, the view controller's view is updated with the new weather data by invoking `updateView()`.

WeekViewController.swift

```
var week: [WeatherDayData]? {
    didSet {
        updateView()
    }
}
```

In `updateView()`, we stop the activity indicator view, stop refreshing the refresh control, and invoke `updateWeatherDataContainer(withWeatherData:)` if there's weather data we need to show the user.

WeekViewController.swift

```
private func updateView() {
    activityIndicatorView.stopAnimating()
    tableView.refreshControl?.endRefreshing()

    if let week = week {
        updateWeatherDataContainer(withWeatherData: week)

    } else {
        messageLabel.isHidden = false
        messageLabel.text = "Cloudy was unable to fetch weather data\
."

    }
}
```

In `updateWeatherDataContainer(withWeatherData:)`, we show the weather data container, which contains the table view, and reload the table view.

WeekViewController.swift

```
private func updateWeatherDataContainer(withWeatherData weatherData:\
 [WeatherDayData]) {
    weatherDataContainer.isHidden = false

    tableView.reloadData()
}
```

The most interesting aspect of the week view controller is the configuration of table view cells in `tableView(_:cellForRowAt:)`. In this method, we dequeue a table view cell, fetch the weather data for the day that corresponds with the index path, and populate the table view cell.

WeekViewController.swift

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: Ind\
exPath) -> UITableViewCell {
    guard let cell = tableView.dequeueReusableCell(withIdentifier: W\
```

```
eatherDayTableViewCell.reuseIdentifier, for: indexPath) as? WeatherD\
ayTableViewCell else { fatalError("Unexpected Table View Cell") }

    if let week = week {
        // Fetch Weather Data
        let weatherData = week[indexPath.row]

        var windSpeed = weatherData.windSpeed
        var temperatureMin = weatherData.temperatureMin
        var temperatureMax = weatherData.temperatureMax

        if UserDefaults.temperatureNotation() != .fahrenheit {
            temperatureMin = temperatureMin.toCelcius()
            temperatureMax = temperatureMax.toCelcius()
        }

        // Configure Cell
        cell.dayLabel.text = dayFormatter.string(from: weatherData.t\
ime)
        cell.dateLabel.text = dateFormatter.string(from: weatherData\
.time)

        let min = String(format: "%.0f°", temperatureMin)
        let max = String(format: "%.0f°", temperatureMax)

        cell.temperatureLabel.text = "\(min) - \(max)"

        if UserDefaults.unitsNotation() != .imperial {
            windSpeed = windSpeed.toKPH()
            cell.windSpeedLabel.text = String(format: "%.f KPH", win\
dSpeed)
        } else {
            cell.windSpeedLabel.text = String(format: "%.f MPH", win\
dSpeed)
        }

        cell.iconImageView.image = imageForIcon(withName: weatherDat\
a.icon)
    }

    return cell
}
```

As in the day view controller, we take the raw values of the model objects and format them before displaying the weather data to the user. Notice that we use several `if` statements to make sure the weather data is formatted based on the user's preferences in the settings view controller.

Locations View Controller

The locations view controller manages a list of locations and it displays the coordinates of the device's current location. If the user selects a location from the list, Cloudy asks the Dark Sky API for that location's weather data and displays it in the weather view controllers.

The user can add a new location by tapping the plus button in the top left. This summons the add location view controller. The user is asked to enter the name of a city. Under the hood, the add location view controller uses the **Core Location** framework to perform a forward geocoding request. Cloudy is only interested in the coordinates of any matches the Core Location framework returns.

Settings View Controller

Despite the simplicity of the settings view, the `SettingsViewController` class is almost 200 lines long. Later in this book, we attempt to use the Model-View-ViewModel pattern to make its implementation shorter and more transparent.

The `SettingsViewController` class has a delegate, which it notifies whenever a setting changes.

SettingsViewController.swift

```
protocol SettingsViewControllerDelegate {
    func controllerDidChangeTimeNotation(controller: SettingsViewCon\
troller)
    func controllerDidChangeUnitsNotation(controller: SettingsViewCo\
ntroller)
    func controllerDidChangeTemperatureNotation(controller: Settings\
ViewController)
}
```

The root view controller is the delegate of the settings view controller and it tells its child view controllers to reload their user interface whenever a setting changes.

RootViewController.swift

```
extension RootViewController: SettingsViewControllerDelegate {

    func controllerDidChangeTimeNotation(controller: SettingsViewCon\
troller) {
        dayViewController.reloadData()
        weekViewController.reloadData()
    }

    func controllerDidChangeUnitsNotation(controller: SettingsViewCo\
ntroller) {
        dayViewController.reloadData()
        weekViewController.reloadData()
    }

```

```
    func controllerDidChangeTemperatureNotation(controller: Settings\
ViewController) {
        dayViewController.reloadData()
        weekViewController.reloadData()
    }

}
```

Time to Write Some Code

That's all you need to know about Cloudy for now. In the next chapter, we focus on several aspects in more detail and discuss which bits we plan to refactor with the help of the Model-View-ViewModel pattern.

If you want to run Cloudy, you need to add your Dark Sky API key to **Configuration.swift**. Signing up for a developer account is free and it only takes a minute.

Configuration.swift

```
struct API {

    static let APIKey = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    static let BaseURL = URL(string: "https://api.darksky.net/foreca\
st/")!

    static var AuthenticatedBaseURL: URL {
        return BaseURL.appendingPathComponent(APIKey)
    }

}
```

4 What Is Wrong With Cloudy

Now that you have an idea of the ins and outs of Cloudy, I'd like to take a few minutes to highlight some of Cloudy's issues. Keep in mind that Cloudy is a small project. The problems we're going to fix with the Model-View-ViewModel pattern are less apparent, which is why I'd like to highlight them before we fix them.

Day View Controller

We start with the day view controller. The first thing to point out is that the view controller keeps a reference to the model. This is a classic example of the Model-View-Controller pattern. Even though there isn't anything inherently wrong with this, when we adopt the Model-View-ViewModel pattern, this will change.

DayViewController.swift

```
1 var now: WeatherData? {
2     didSet {
3         updateView()
4     }
5 }
```

The second and most important problem is the implementation of the `updateWeatherDataContainer(withWeatherData:)` method. This is another pattern that's typical for the Model-View-Controller pattern. The raw values of the model data are transformed and formatted before they're displayed to the user.

DayViewController.swift

```
1 private func updateWeatherDataContainer(withWeatherData weatherData:\
2  WeatherData) {
3     weatherDataContainer.isHidden = false
4 
5     var windSpeed = weatherData.windSpeed
6     var temperature = weatherData.temperature
7 
8     let dateFormatter = DateFormatter()
9     dateFormatter.dateFormat = "EEE, MMMM d"
```

```
10      dateLabel.text = dateFormatter.string(from: weatherData.time)
11
12      let timeFormatter = DateFormatter()
13
14      if UserDefaults.timeNotation() == .twelveHour {
15          timeFormatter.dateFormat = "hh:mm a"
16      } else {
17          timeFormatter.dateFormat = "HH:mm"
18      }
19
20      timeLabel.text = timeFormatter.string(from: weatherData.time)
21
22      descriptionLabel.text = weatherData.summary
23
24      if UserDefaults.temperatureNotation() != .fahrenheit {
25          temperature = temperature.toCelcius()
26          temperatureLabel.text = String(format: "%.1f °C", temperatur\
27  e)
28      } else {
29          temperatureLabel.text = String(format: "%.1f °F", temperatur\
30  e)
31      }
32
33      if UserDefaults.unitsNotation() != .imperial {
34          windSpeed = windSpeed.toKPH()
35          windSpeedLabel.text = String(format: "%.f KPH", windSpeed)
36      } else {
37          windSpeedLabel.text = String(format: "%.f MPH", windSpeed)
38      }
39
40      iconImageView.image = imageForIcon(withName: weatherData.icon)
41  }
```

Should the view controller be in charge of this task? Maybe. Maybe not. But is there a more elegant solution? Absolutely.

If we adopt the Model-View-ViewModel pattern, the view controller will no longer be responsible for data manipulation. Moreover, the view controller won't know about and have direct access to the model. It will receive a view model from the root view controller and use the view model to populate its view. That's the task it was designed for, controlling a view.

Week View Controller

The week view controller suffers from the same problems. It keeps a strong reference to the array of `WeatherDayData` objects and uses them to populate the table view.

WeekViewController.swift

```
var week: [WeatherDayData]? {
    didSet {
        updateView()
    }
}
```

In the `tableView(cellForRowAt:)` method, a `WeatherDayData` instance is fetched from the array and it's used to populate a table view cell. The raw values of the model data are transformed and formatted before they're displayed to the user.

WeekViewController.swift

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: Ind\
exPath) -> UITableViewCell {
    guard let cell = tableView.dequeueReusableCell(withIdentifier: W\
eatherDayTableViewCell.reuseIdentifier, for: indexPath) as? WeatherD\
ayTableViewCell else { fatalError("Unexpected Table View Cell") }

    if let week = week {
        // Fetch Weather Data
        let weatherData = week[indexPath.row]

        var windSpeed = weatherData.windSpeed
        var temperatureMin = weatherData.temperatureMin
        var temperatureMax = weatherData.temperatureMax

        if UserDefaults.temperatureNotation() != .fahrenheit {
            temperatureMin = temperatureMin.toCelcius()
            temperatureMax = temperatureMax.toCelcius()
        }

        // Configure Cell
        cell.dayLabel.text = dayFormatter.string(from: weatherData.t\
ime)
        cell.dateLabel.text = dateFormatter.string(from: weatherData\
.time)

        let min = String(format: "%.0f°", temperatureMin)
        let max = String(format: "%.0f°", temperatureMax)

        cell.temperatureLabel.text = "\(min) - \(max)"

        if UserDefaults.unitsNotation() != .imperial {
            windSpeed = windSpeed.toKPH()
            cell.windSpeedLabel.text = String(format: "%.f KPH", win\
dSpeed)
        } else {
            cell.windSpeedLabel.text = String(format: "%.f MPH", win\
dSpeed)
        }

        cell.iconImageView.image = imageForIcon(withName: weatherDat\
a.icon)
    }

    return cell
}
```

We also see several `if` statements to make sure the raw values are formatted correctly, based on the user's preferences.

The Model-View-Controller pattern has a few other consequences. The week view controller has a couple of properties of type `DateFormatter` to format the model data that's displayed in the table view. If we use the Model-View-ViewModel pattern, we can clean this up too. Whenever I see a `DateFormatter` property in a view controller, I know it's time for some refactoring.

Locations View Controller

Later in this book, we focus on the locations view controller. It will show you how user interaction is handled by the Model-View-ViewModel pattern. This is a bit more complicated. However, once you understand the ins and outs of the Model-View-ViewModel pattern, this won't be difficult to understand. I promise you that the result is pure elegance.

Settings View Controller

There doesn't seem to be anything wrong with the settings view controller. It's true that it doesn't look too bad, but I assure you that it'll look a lot better after we've given the settings view controller a facelift using protocols and MVVM.

What's Next

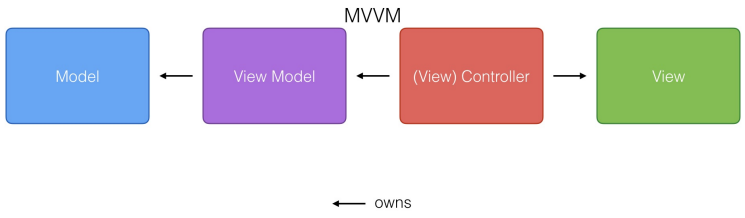
In the next chapters, you create your very first view model. We start with the view model for the day view controller.

5 A Quick Recap

MVVM Architecture

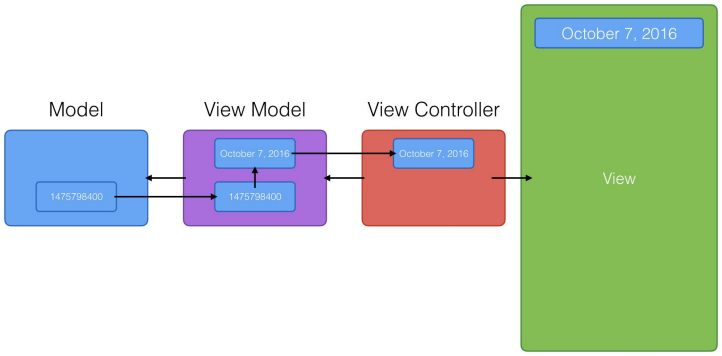
Before you create your first view model, I want to revisit the internals of the Model-View-ViewModel pattern. We need to keep the following in mind:

- the model is owned by the view model
- the view controller doesn't know about and cannot access the model
- the view controller owns the view model
- the view model doesn't know about the view controller it's owned by



MVVM Architecture

These are the four key elements you need to remember for this and the next chapters. Let me show you what happens when the view controller needs to display a piece data in the view it manages.



An Example

The view controller asks its view model for a piece of data. The view model asks the model it manages for the raw value, a timestamp for example. The view model applies the necessary transformations to the raw value and returns a value the view controller can immediately display to the user in its view.

That’s the flow we will see time and time again in the next chapters. The view controller is no longer responsible for transforming the raw values of the model and it doesnâ€™t even know about the model. That’s an important difference with the Model-View-Controller pattern.

Naming the View Model

A common question is “How should I name the view model?” There are several opinions and the best advice is “Do what feels right and makes sense. And, more importantly, make sure you understand the code you write today a year from now.”

The view models I create are very often focused on a specific view controller. For example, we’ll create a view model for the day view controller and a view model for the week view controller. Because the view model is linked to the view of a view controller, I name the view model after the view it’s used for.

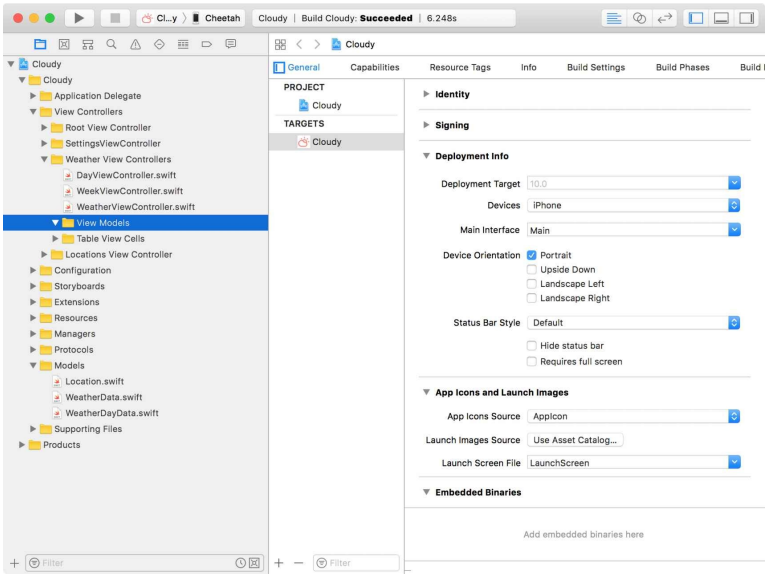
[View]ViewModel

For example, we'll name the view model for the day view controller **DayViewViewModel**. You're free to choose the name of your view models, but I recommend that you use the **ViewModel** suffix to clearly indicate that you're dealing with a view model.

6 Time to Create a View Model

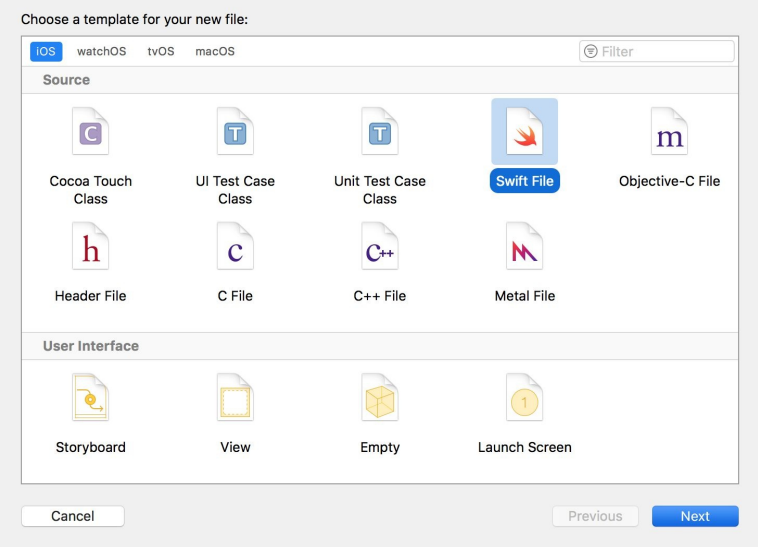
Creating the View Model

In this chapter, we create a view model for the day view controller. Open Cloudy in Xcode and create a new group, **View Models**, in the **Weather View Controllers** group. I prefer to keep the view models close to the view controllers in which they're used.

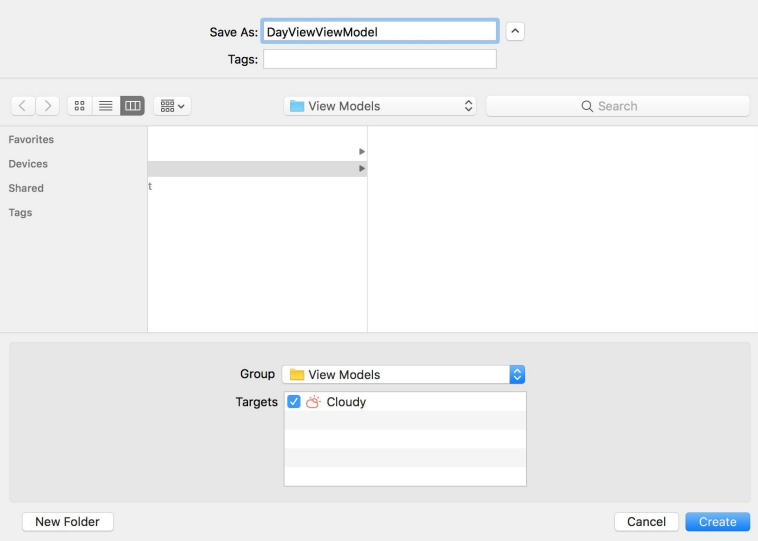


Creating the View Models Group

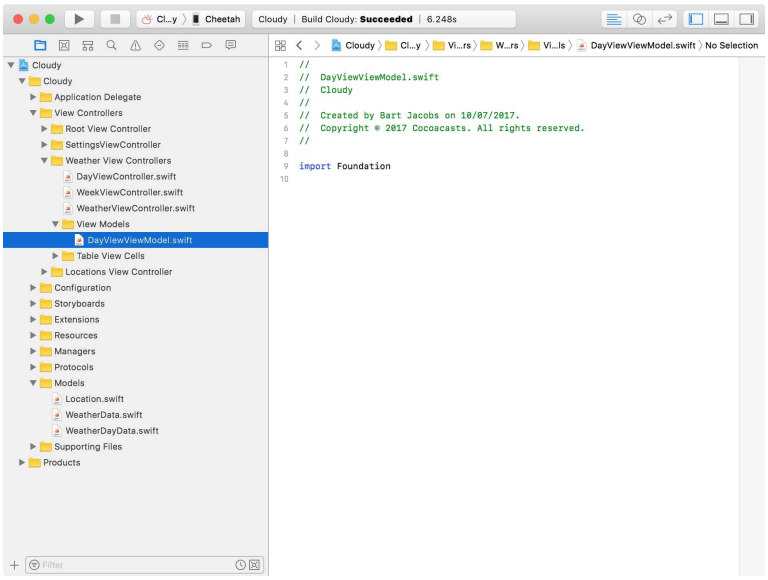
Create a new Swift file in the **View Models** group and name it **DayViewViewModel**.



Creating the Day View View Model



Creating the Day View View Model



Creating the Day View View Model

`DayViewViewModel` is a struct, a **value type**. Remember that the view model should keep a reference to the model, which means we need to create a property for it. That's all we need to do to create our first view model.

DayViewViewModel.swift

```
import Foundation

struct DayViewViewModel {

    // MARK: - Properties

    let weatherData: WeatherData

}
```

Creating the Public Interface

The next step is moving the code located in the `updateWeatherDataContainer(withWeatherData:)` method of the `DayViewController` class to the view model. What we need to focus on are the values we use to populate the user interface.

Date Label

Let's start with the date label. The date label expects a formatted date and it needs to be of type `String`. It's the responsibility of the view model

to ask the model for the value of its `time` property and transform that value to the format the date label expects.

Let's start by creating a computed property in the view model. We name it `date` and it should be of type `String`.

DayViewViewModel.swift

```
var date: String {

}
```

We initialize a `DateFormatter` to convert the date to a formatted string and set the date formatter's `dateFormat` property. We invoke the date formatter's `string(from:)` method and return the result. That's it for the date label. This is what the `date` computed property looks like.

DayViewViewModel.swift

```
var date: String {
    // Initialize Date Formatter
    let dateFormatter = DateFormatter()

    // Configure Date Formatter
    dateFormatter.dateFormat = "EEE, MMMM d"

    return dateFormatter.string(from: weatherData.time)
}
```

Time Label

We can repeat this for the time label. We first create a `time` computed property of type `String`. The implementation is similar. We create a `DateFormatter` instance, set its `dateFormat` property, and return a formatted string.

DayViewViewModel.swift

```
var time: String {
    // Initialize Date Formatter
    let dateFormatter = DateFormatter()

    // Configure Date Formatter
    dateFormatter.dateFormat = ""

    return dateFormatter.string(from: weatherData.time)
}
```

There's one complication, though. The format of the time depends on the user's settings in the application. This is easy to solve, though. Navigate to **UserDefaults.swift** in the **Extensions** group. We add a computed property, `timeFormat`, to the `TimeNotation` enum at the top. The `timeFormat` computed property returns the correct date format based on the user's preferences. This is what that looks like.

UserDefaults.swift

```
enum TimeNotation: Int {
    case twelveHour
    case twentyFourHour

    var timeFormat: String {
        switch self {
        case .twelveHour: return "hh:mm a"
        case .twentyFourHour: return "HH:mm"
        }
    }
}
```

We can now update the implementation of the `time` computed property like this.

DayViewViewModel.swift

```
var time: String {
    // Initialize Date Formatter
    let dateFormatter = DateFormatter()

    // Configure Date Formatter
    dateFormatter.dateFormat = UserDefaults.timeNotation().timeFormat

    return dateFormatter.string(from: weatherData.time)
}
```

Confused? The `timeNotation` method is a static method of the `UserDefaults` class. You can find its implementation in **UserDefaults.swift**. It returns a `TimeNotation` instance. Take a look at the implementation.

UserDefaults.swift

```
static func timeNotation() -> TimeNotation {
    let storedValue = UserDefaults.standard.integer(forKey: UserDefa\
ultsKeys.timeNotation)
    return TimeNotation(rawValue: storedValue) ?? TimeNotation.twelv\
eHour
}
```

We load the user's preference from the user defaults database and use the value to create an instance of the `TimeNotation` enum. We use the same technique for the user's other preferences.

Description Label

Populating the description label is easy. We define a computed property in the view model, `summary`, of type `String` and we return the value of the `summary` property of the model.

DayViewViewModel.swift

```
var summary: String {
    return weatherData.summary
}
```

Temperature Label

The value for the temperature label is a bit more complicated because we need to take the user's preferences into account. We start simple. We create another computed property in which we store the temperature in a constant, `temperature`.

DayViewViewModel.swift

```
var temperature: String {
    let temperature = weatherData.temperature
}
```

We fetch the user's preference and format the value stored in the `temperature` constant based on the user's preference. Notice that we need to convert the temperature if the user's preference is set to degrees Celcius.

DayViewViewModel.swift

```
var temperature: String {
    let temperature = weatherData.temperature

    switch UserDefaults.temperatureNotation() {
    case .fahrenheit:
        return String(format: "%.1f F", temperature)
    case .celsius:
        return String(format: "%.1f C", temperature.toCelcius())
    }
}
```

The implementation of the `temperatureNotation()` static method is very similar to the `timeNotation()` static method we looked at earlier.

UserDefaults.swift

```
static func temperatureNotation() -> TemperatureNotation {
    let storedValue = UserDefaults.standard.integer(forKey: UserDefa\
ultsKeys.temperatureNotation)
    return TemperatureNotation(rawValue: storedValue) ?? Temperature\
Notation.fahrenheit
}
```

Wind Speed Label

Populating the wind speed label is very similar. Because the wind speed label expects a string, we create a `windSpeed` computed property of type `String`. We ask the model for the the value of its `windSpeed` property and format that value based on the user's preference.

DayViewViewModel.swift

```
var windSpeed: String {
    let windSpeed = weatherData.windSpeed

    switch UserDefaults.unitsNotation() {
    case .imperial:
        return String(format: "%.f MPH", windSpeed)
    case .metric:
        return String(format: "%.f KPH", windSpeed.toKPH())
    }
}
```

The implementation of the `unitsNotation()` static method is very similar to the `timeNotation()` and `temperatureNotation()` static methods we looked at earlier.

UserDefaults.swift

```
static func unitsNotation() -> UnitsNotation {
    let storedValue = UserDefaults.standard.integer(forKey: UserDefa\
ultsKeys.unitsNotation)
    return UnitsNotation(rawValue: storedValue) ?? UnitsNotation.imp\
erial
}
```

Icon Image View

For the icon image view, we need an image. We could put this logic in the view model. However, because we need the same logic later, in the view model of the week view controller, it's better to create an extension for `UIImage` in which we put that logic.

Create a new file in the **Extensions** group and name it **UIImage.swift**. Create an extension for the `UIImage` class and define a class method `imageForIcon(withName:)`.

UIImage.swift

```
1 import UIKit
2
3 extension UIImage {
4
5     class func imageForIcon(withName name: String) -> UIImage? {
6
7     }
8
9 }
```

We simplify the current implementation of the weather view controller. We use the value of the `name` argument to instantiate the `UIImage` instance in most cases of the `switch` statement. I really like how flexible the `switch` statement is in Swift.

UIImage.swift

```
 1 import UIKit
 2
 3 extension UIImage {
 4
 5     class func imageForIcon(withName name: String) -> UIImage? {
 6         switch name {
 7         case "clear-day", "clear-night", "rain", "snow", "sleet": re\
 8 turn UIImage(named: name)
 9         case "wind", "cloudy", "partly-cloudy-day", "partly-cloudy-n\
10 ight": return UIImage(named: "cloudy")
11         default: return UIImage(named: "clear-day")
12         }
13     }
14
15 }
```

Notice that we also return a `UIImage` instance in the `default` case of the `switch` statement.

With this method in place, it's very easy to populate the icon image view. We create a computed property of type `UIImage?` in the view model and name it `image`. In the body of the computed property, we invoke the class method we just created, passing in the value of the model's `icon` property.

DayViewViewModel.swift

```
var image: UIImage? {
    return UIImage.imageForIcon(withName: weatherData.icon)
}
```

Because `UIImage` is defined in the **UIKit** framework, we need to replace the import statement for **Foundation** with an import statement for **UIKit**.

DayViewViewModel.swift

```
import UIKit

struct DayViewViewModel {

    ...

}
```

This is a code smell. Whenever you import **UIKit** in a view model, a warning bell should go off. The view model shouldn't need to know anything about views or the user interface. In this example, however, we have no other option. Since we want to return a `UIImage` instance, we need to import **UIKit**. If you don't like this, you can also return the name of the image and have the view controller be in charge of creating the `UIImage` instance. That's up to you.

We're almost done. I want to make two small improvements. The `DateFormatter` instance shouldn't be created in the computed properties in my opinion. We can create private properties for those.

DayViewViewModel.swift

```
import UIKit

struct DayViewViewModel {

    // MARK: - Properties

```

```
    let weatherData: WeatherData

    // MARK: -

    private let dateFormatter = DateFormatter()
    private let timeFormatter = DateFormatter()

    // MARK: -

    var date: String {
        // Configure Date Formatter
        dateFormatter.dateFormat = "EEE, MMMM d"

        return dateFormatter.string(from: weatherData.time)
    }

    var time: String {
        // Configure Date Formatter
        timeFormatter.dateFormat = UserDefaults.timeNotation().timeF\
ormat

        return timeFormatter.string(from: weatherData.time)
    }

    ...

}
```

I've chosen to create a separate date formatter for the `date` and `time` properties. You could use the same date formatter for both properties and only update the date formatter's `dateFormat` property. It's up to you to decide which option you like most.

Great. We've created our first view model. In the next chapter, we put it to use in the day view controller.

7 Put the View Model to Work

If you're not sure how the various pieces of the Model-View-ViewModel pattern fit together, then this chapter is certainly going to help. In this chapter, we put the `DayViewViewModel` we created in the previous chapter to work. This means we need to refactor the day view controller and the root view controller.

Adding the View Model to the Day View Controller

The first step we need to take is replacing the `now` property of the day view controller with an instance of the `DayViewViewModel` struct. Only the name and the type of the property change. We can leave the `didSet` property observer untouched.

DayViewController.swift

```
var viewModel: DayViewViewModel? {
    didSet {
        updateView()
    }
}
```

Creating the View Model in the Root View Controller

With the `now` property removed from the `DayViewController` class, we need to update the root view controller. It no longer passes the `WeatherData` instance to the day view controller. Instead, the root view controller instantiates an instance of the `DayViewViewModel` using the `WeatherData` instance and sets the `viewModel` property of the day view controller.

Open **RootViewController.swift** and navigate to the `fetchWeatherData()` method. In the completion handler of the `weatherDataForLocation(latitude:longitude:completion:)` method, we create an instance of the `DayViewViewModel` struct and assign it to the `viewModel` property of the day view controller.

RootViewController.swift

```
private func fetchWeatherData() {
    ...

    dataManager.weatherDataForLocation(latitude: latitude, longitude\
: longitude) { (response, error) in
        if let error = error {
            print(error)
        } else if let response = response {
            // Configure Day View Controller
            self.dayViewController.viewModel = DayViewViewModel(weat\
herData: response)

            // Configure Week View Controller
            self.weekViewController.week = response.dailyData
        }
    }
}
```

This confirms what we discussed in the previous chapter. The day view controller no longer has direct access to the `WeatherData` instance. It can only access the model indirectly through the view model.

Updating the Weather Data Container

The last piece of the puzzle is updating the `updateWeatherDataContainer(withWeatherData:)` method of the `DayViewController` class. But first, we're going to rename the method to `updateWeatherDataContainer(withViewModel:)`. That makes more sense.

DayViewController.swift

```
private func updateWeatherDataContainer(withViewModel viewModel: Day\
ViewViewModel) {
      ...
}
```

This also means we need to update the `updateView()` method of the `DayViewController` class. Notice that I've also replaced any references to the `now` property with references to the `viewModel` property.

DayViewController.swift

```
private func updateView() {
    activityIndicatorView.stopAnimating()

    if let viewModel = viewModel {
        updateWeatherDataContainer(withViewModel: viewModel)

```

```
    } else {
        ...
    }
}
```

The implementation of `updateWeatherDataContainer(withViewModel:)` becomes much shorter. Instead of using the raw values of the model, we simply ask the view model for the data the view controller needs to display in its view.

DayViewController.swift

```
private func updateWeatherDataContainer(withViewModel viewModel: Day\
ViewViewModel) {
    weatherDataContainer.isHidden = false

    dateLabel.text = viewModel.date
    timeLabel.text = viewModel.time
    iconImageView.image = viewModel.image
    windSpeedLabel.text = viewModel.windSpeed
    descriptionLabel.text = viewModel.summary
    temperatureLabel.text = viewModel.temperature
}
```

We're benefiting from the work we did in the previous chapter. To populate the `dateLabel`, we ask the view model for its `date` property, the computed property we implemented in the previous chapter. We update the `timeLabel` with the value of the `time` property of the view model, *etc.* The resulting implementation is short, clear, and focused.

I'm sure you agree that deleting code is one of the more enjoyable aspects of software development. The day view controller is now much leaner and very focused. It presents data and responds to user interaction. That's what view controllers are designed for.

Build and Run

Run the application to see if everything's still working. You'll notice that nothing has changed. If something doesn't look right, then you probably made a mistake in the view model.

Even though the day view controller displays a simple user interface, I hope you now understand how the Model-View-ViewModel pattern helps to keep your project's view controllers in check. The Model-View-ViewModel pattern delegates tasks and responsibilities of the view

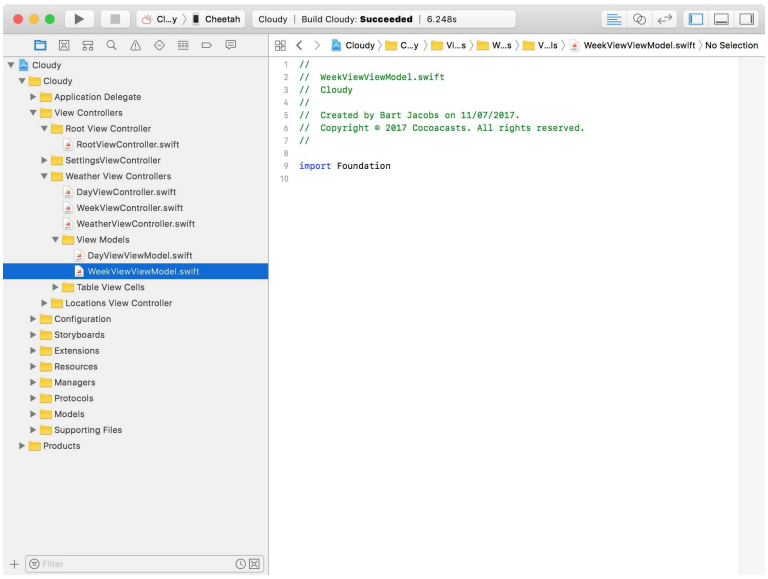
controller to the view model.

I’ve been using the Model-View-ViewModel pattern in more complex projects and the results are pretty impressive. View controllers no longer tend to be massive and the testability of your project improves dramatically. We’re only scratching the surface, though. In the next chapter, we shift focus to the week view controller.

8 Rinse and Repeat

Creating the Week View View Model

To adopt the Model-View-ViewModel pattern in the `WeekViewController` class, we first need to create a new view model. Create a new file in the **View Models** group and name the file **WeekViewViewModel.swift**.



Creating the Week View View Model

Replace the import statement for **Foundation** with an import statement for **UIKit** and declare the `WeekViewViewModel` struct.

WeekViewViewModel.swift

```
1 import UIKit
2
3 struct WeekViewViewModel {
4
5 }
```

You already know what the next step is. We need to create a property for the weather data the view model manages. We name the property `weatherData` and the property is of type array of `WeatherDayData` instances.

WeekViewViewModel.swift

```
import UIKit

struct WeekViewViewModel {

    // MARK: - Properties

    let weatherData: [WeatherDayData]

}
```

The `WeekViewViewModel` struct will look a bit different from the `DayViewViewModel` struct because it manages an array of model objects. Once we're done refactoring the `WeekViewController` class, it will no longer have a reference to the weather data. This means that the week view controller won't know how many sections and rows the table view should display. That information needs to come from the view controller's view model.

Let's start by adding a new computed property to the `WeekViewViewModel` struct, `numberOfSections`. The `numberOfSections` computed property returns the number of sections the week view controller should display in the table view it manages. The view controller currently display only one section, which means we return `1`.

WeekViewViewModel.swift

```
var numberOfSections: Int {
    return 1
}
```

The view controller also needs to know how many rows that section contains. To answer this question, we add another computed property, `numberOfDays`, which tells the view controller the number of days the view model has weather data for. We could implement a fancy method that accepts an index corresponding with the section in the table view, but I prefer to keep view models as simple and as dumb as possible. The view controller and its view model should only be given information they absolutely need to carry out their tasks.

WeekViewViewModel.swift

```
var numberOfDays: Int {
    return weatherData.count
}
```

Up until now, we used computed properties to provide the view controller with the information it needs. It's time for a few methods. These methods will provide the view controller with weather data for a particular day. This means that the view controller asks the view model for the weather data for a specific index. The index corresponds with a row in the table view.

Let's start with the day label of the `WeatherDayTableViewCell`. The view controller will ask its view model for a string that represents the day of the weather data. This means we need to implement a method in the `WeekViewViewModel` struct that accepts an index of type `Int` and returns a value of type `String`.

WeekViewViewModel.swift

```
func day(for index: Int) -> String {

}
```

We first fetch the `WeatherDayData` instance that corresponds with the index that's passed to the `day(index:)` method. We create a `DateFormatter` instance and format the value of the `time` property of the model.

WeekViewViewModel.swift

```
func day(for index: Int) -> String {
    // Fetch Weather Data for Day
    let weatherDayData = weatherData[index]

    // Initialize Date Formatter
    let dateFormatter = DateFormatter()

    // Configure Date Formatter
    dateFormatter.dateFormat = "EEEE"

    return dateFormatter.string(from: weatherDayData.time)
}
```

As I mentioned earlier in this book, I prefer to make the `DateFormatter` instance a property of the view model. Let's take care of that now. This is what the `WeekViewViewModel` struct looks like so far.

WeekViewViewModel.swift

```
import UIKit

struct WeekViewViewModel {

    // MARK: - Properties

    let weatherData: [WeatherDayData]

    // MARK: -

    private let dayFormatter = DateFormatter()

    // MARK: -

    var numberOfSections: Int {
        return 1
    }

    // MARK: - Methods

    func day(for index: Int) -> String {
        // Fetch Weather Data for Day
        let weatherDayData = weatherData[index]

        // Configure Date Formatter
        dayFormatter.dateFormat = "EEEE"

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

}
```

We can use the same approach to populate the date label of the `WeatherDayTableViewCell`. Only the value of the date formatter's `dateFormat` property is different. You could argue that we could get away with only one `DateFormatter` property. That's a personal choice. It won't dramatically impact performance since we only have a handful of table view cells to populate.

WeekViewViewModel.swift

```
import UIKit

struct WeekViewViewModel {

    // MARK: - Properties

    let weatherData: [WeatherDayData]

    // MARK: -

    private let dayFormatter = DateFormatter()
    private let dateFormatter = DateFormatter()

```

```
14      ...
15
16      func date(for index: Int) -> String {
17          // Fetch Weather Data for Day
18          let weatherDayData = weatherData[index]
19
20          // Configure Date Formatter
21          dateFormatter.dateFormat = "MMMM d"
22
23          return dateFormatter.string(from: weatherDayData.time)
24      }
25
26  }
```

Setting the `text` property of the temperature label of the `WeatherDayTableViewCell` is another fine example of the elegance and versatility of view models. Remember that the `WeatherDayTableViewCell` displays the minimum and the maximum temperature for a particular day. The view model should provide the formatted string to the view controller so that it can pass it to the `WeatherDayTableViewCell`.

In the `temperature(for:)` method, we fetch the weather data, format the minimum and maximum temperatures using a helper method, `format(temperature:)`, and return the formatted string as a result.

WeekViewViewModel.swift

```
1 func temperature(for index: Int) -> String {
2     // Fetch Weather Data
3     let weatherDayData = weatherData[index]
4
5     let min = format(temperature: weatherDayData.temperatureMin)
6     let max = format(temperature: weatherDayData.temperatureMax)
7
8     return "\(min) - \(max)"
9 }
```

The `format(temperature:)` helper method isn't complicated. It only prevents us from repeating ourselves.

WeekViewViewModel.swift

```
1 // MARK: - Helper Methods
2
3 private func format(temperature: Double) -> String {
4     switch UserDefaults.temperatureNotation() {
5     case .fahrenheit:
6         return String(format: "%.0f F", temperature)
7     case .celsius:
8         return String(format: "%.0f C", temperature.toCelcius())
```

```
    }
}
```

Populating the wind speed label and the icon image view is very similar to what we covered so far. Take a look at the implementations below.

WeekViewViewModel.swift

```
func windSpeed(for index: Int) -> String {
    // Fetch Weather Data
    let weatherDayData = weatherData[index]
    let windSpeed = weatherDayData.windSpeed

    switch UserDefaults.unitsNotation() {
    case .imperial:
        return String(format: "%.f MPH", windSpeed)
    case .metric:
        return String(format: "%.f KPH", windSpeed.toKPH())
    }
}
```

WeekViewViewModel.swift

```
func image(for index: Int) -> UIImage? {
    // Fetch Weather Data
    let weatherDayData = weatherData[index]

    return UIImage.imageForIcon(withName: weatherDayData.icon)
}
```

Creating the View Model in the Root View Controller

With the `WeekViewViewModel` struct ready to use, we shift focus to the `RootViewController` class. First, however, we replace the `week` property in the `WeekViewController` class with a property named `viewModel` of type `WeekViewViewModel?`.

WeekViewController.swift

```
var viewModel: WeekViewViewModel? {
    didSet {
        updateView()
    }
}
```

We can now update the `RootViewController` class. The root view controller no longer passes the array of `WeatherDayData` instances to the week view controller. Instead, the root view controller instantiates an

instance of the `WeekViewViewModel` using the array of `WeatherDayData` instances and sets the `viewModel` property of the week view controller.

Open **RootViewController.swift** and navigate to the `fetchWeatherData()` method. In the completion handler of the `weatherDataForLocation(latitude:longitude:completion:)` method, we create an instance of the `WeekViewViewModel` struct and assign it to the `viewModel` property of the week view controller.

RootViewController.swift

```
private func fetchWeatherData() {
    ...

    dataManager.weatherDataForLocation(latitude: latitude, longitude\
: longitude) { (response, error) in
        if let error = error {
            print(error)
        } else if let response = response {
            // Configure Day View Controller
            self.dayViewController.viewModel = DayViewViewModel(weat\
herData: response)

            // Configure Week View Controller
            self.weekViewController.viewModel = WeekViewViewModel(we\
atherData:  response.dailyData)
        }
    }
}
```

Updating the Table View

Revisit **WeekViewController.swift**. With the `WeekViewViewModel` struct ready to use, it's time to refactor the `WeekViewController` class. This means we need to update the `updateWeatherDataContainer(withWeatherData:)` method. We start by renaming this method to `updateWeatherDataContainer()`. There's no need to pass in the view model like we did in the `DayViewController` class.

WeekViewController.swift

```
private func updateWeatherDataContainer() {
    weatherDataContainer.isHidden = false

    tableView.reloadData()
}
```

We also update the `updateView()` method to reflect these changes. We also replace any references to the `week` property with references to the `viewModel` property.

WeekViewController.swift

```
private func updateView() {
    activityIndicatorView.stopAnimating()
    tableView.refreshControl?.endRefreshing()

    if let _ = viewModel {
        updateWeatherDataContainer()

    } else {
     ...
    }
}
```

The implementation of the `UITableViewDataSource` protocol also needs some changes. As you can see, we use the methods we implemented earlier in the `WeekViewViewModel` struct.

WeekViewController.swift

```
func numberOfSections(in tableView: UITableView) -> Int {
    guard let viewModel = viewModel else { return 0 }
    return viewModel.numberOfSections
}
```

WeekViewController.swift

```
func tableView(_ tableView: UITableView, numberOfRowsInSection secti\
on: Int) -> Int {
    guard let viewModel = viewModel else { return 0 }
    return viewModel.numberOfDays
}
```

Thanks to the `WeekViewViewModel` struct, we can greatly simplify the implementation of `tableView(_:cellForRowAt:)`.

WeekViewController.swift

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: Ind\
exPath) -> UITableViewCell {
    guard let cell = tableView.dequeueReusableCell(withIdentifier: W\
eatherDayTableViewCell.reuseIdentifier, for: indexPath) as? WeatherD\
ayTableViewCell else { fatalError("Unexpected Table View Cell") }

    if let viewModel = viewModel {
        // Configure Cell
```

```
        cell.dayLabel.text = viewModel.day(for: indexPath.row)
        cell.dateLabel.text = viewModel.date(for: indexPath.row)
        cell.iconImageView.image = viewModel.image(for: indexPath.ro\
w)
        cell.windSpeedLabel.text = viewModel.windSpeed(for: indexPat\
h.row)
        cell.temperatureLabel.text = viewModel.temperature(for: inde\
xPath.row)
    }

    return cell
}
```

Last but not least, we can get rid of the `DateFormatter` properties of the `WeekViewController`. They're no longer needed and that's a very welcome change.

Even though we successfully implemented the Model-View-ViewModel pattern in the week view controller, later in this book, we use protocols to further simplify the implementation of the Model-View-ViewModel pattern in the week view controller.

9 Using MVVM In the Settings View

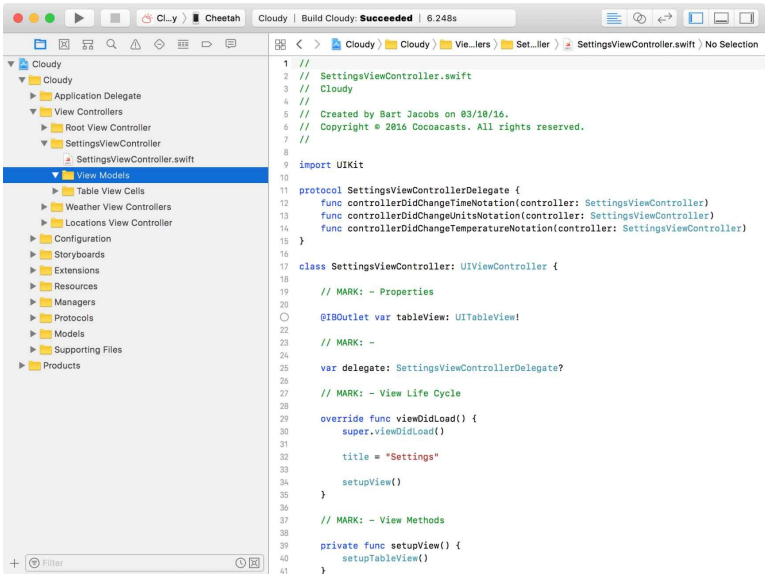
The Model-View-ViewModel pattern isn't only useful for populating data-driven user interfaces. In this chapter, I want to show you how you can apply the Model-View-ViewModel pattern in the settings view controller.

Remember that we want to extract logic from the view controller that doesn't belong there. In this chapter, I'd like to target the `tableView(_:cellForRowAt:)` method of the settings view controller. Instead of figuring out what the view controller should display, we're going to create three view models the view controller can use to populate the table view.

This example also illustrates that view models are sometimes short-lived objects. The view controller doesn't necessarily need to keep a reference to the view model. That's something we haven't covered yet in this book.

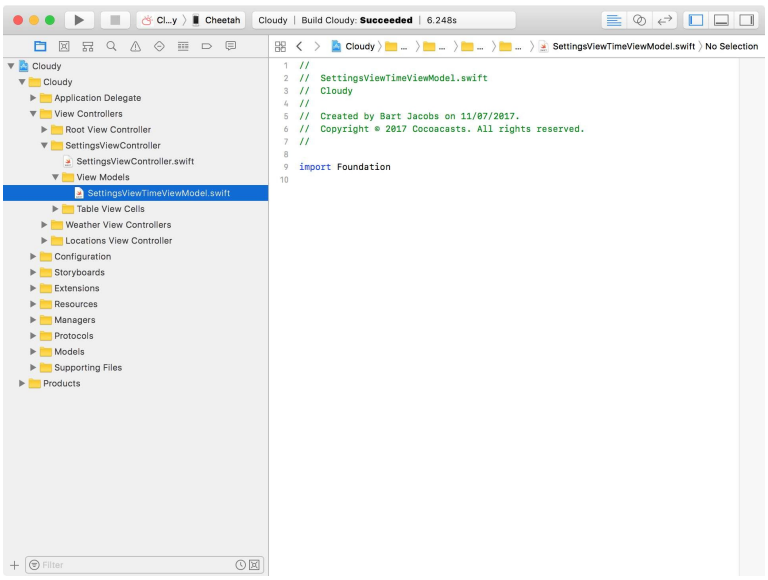
Creating the Settings View Time View Model

We start by creating a new group for the view models.



Creating the View Models Group

The first view model we create is a view model for populating the table view cells of the first section of the table view, the time section. This section is used for switching between 12 hour and 24 hour time notation. We create a new file and name it **SettingsViewTimeViewModel.swift**.



Creating SettingsViewTimeViewModel.swift

We replace the import statement for **Foundation** with an import statement for **UIKit** and define the `SettingsViewTimeViewModel` struct.

SettingsViewTimeViewModel.swift

```
1 import UIKit
2
3 struct SettingsViewTimeViewModel {
4
5 }
```

The model of the struct should be of type `TimeNotation`. We define a new property, `timeNotation`, of type `TimeNotation`.

SettingsViewTimeViewModel.swift

```
1 import UIKit
2
3 struct SettingsViewTimeViewModel {
4
5     // MARK: - Properties
6
7     let timeNotation: TimeNotation
8
9 }
```

If we jump back to the implementation of `tableView(_:cellForRowAt:)` in the `SettingsViewController` class, we can see which values the view model should provide the view controller with. The view controller needs a value for the main label of the table view cell and it also needs to set the `accessoryType` property of the table view cell.

SettingsViewController.swift

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: Ind\
exPath) -> UITableViewCell {
    guard let section = Section(rawValue: indexPath.section) else { \
fatalError("Unexpected Section") }
    guard let cell = tableView.dequeueReusableCell(withIdentifier: S\
ettingsTableViewCell.reuseIdentifier, for: indexPath) as? SettingsTa\
bleViewCell else { fatalError("Unexpected Table View Cell") }

    switch section {
    case .time:
        cell.mainLabel.text = (indexPath.row == 0) ? "12 Hour" : "24\
 Hour"

        let timeNotation = UserDefaults.timeNotation()
        if indexPath.row == timeNotation.rawValue {
            cell.accessoryType = .checkmark
        } else {
            cell.accessoryType = .none
        }
    case .units:
        ...
    case .temperature:
        ...
    }

    return cell
}
```

This means we need to implement two computed properties. The first computed property is named `text` and it's of type `String`. We use a simple `switch` statement to return a string based on the value of the `timeNotation` property. This should look familiar by now.

SettingsViewTimeViewModel.swift

```
var text: String {
    switch timeNotation {
    case .twelveHour: return "12 Hour"
    case .twentyFourHour: return "24 Hour"
    }
}
```

We create another computed property, `accessoryType`, of type `UITableViewCellAccessoryType`. We fetch the user's preference from the

user defaults database and compare it with the value of the `timeNotation` property. The result of this comparison determines the accessory type we return.

SettingsViewTimeViewModel.swift

```
var accessoryType: UITableViewCellAccessoryType {
    if UserDefaults.timeNotation() == timeNotation {
        return .checkmark
    } else {
        return .none
    }
}
```

That's all we need to do to implement the `SettingsViewTimeViewModel` struct.

Importing UIKit

Earlier in this book, I mentioned that importing the **UIKit** framework in a view model is a code smell. It's not necessarily wrong, but you need to be careful.

What we're doing in the `SettingsViewTimeViewModel` struct is open for discussion. Some developers will argue that this is wrong because we reference a trait of a view, a table view cell. I can understand and appreciate that argument.

No matter how you feel about the implementation of the `SettingsViewTimeViewModel` struct, what's most important is that you're aware of what we're doing. I have a clear philosophy about this subject. It's fine to break a rule as long as you know what the rule stands for and you understand the consequences.

That's enough philosophy for now. It's time to put the `SettingsViewTimeViewModel` struct to work in the `SettingsViewController` class.

Refactoring the Settings View Controller

We can now use the `SettingsViewTimeViewModel` struct in the `SettingsViewController` class. Open **SettingsViewController.swift** and

navigate to the `tableView(_:cellForRowAt:)` method. In the `switch` statement, we initialize an instance of the `TimeNotation` enum using the value of the `indexPath` argument.

SettingsViewController.swift

```
case .time:
    guard let timeNotation = TimeNotation(rawValue: indexPath.row) e\
lse {
        fatalError("Unexpected Index Path")
    }
```

We use the `timeNotation` instance to create the view model.

SettingsViewController.swift

```
// Initialize View Model
let viewModel = SettingsViewTimeViewModel(timeNotation: timeNotation)
```

We use the view model to configure the table view cell.

SettingsViewController.swift

```
// Configure Cell
cell.mainLabel.text = viewModel.text
cell.accessoryType = viewModel.accessoryType
```

Your Turn

You should now be able to create the view models for the remaining two sections. Put the book aside for a moment and give it a try.

This is what the `SettingsViewUnitsViewModel` struct should look like. The most important difference with the `SettingsViewTimeViewModel` struct is the name and type of the model the view model manages.

SettingsViewUnitsViewModel.swift

```
import UIKit

struct SettingsViewUnitsViewModel {

    // MARK: - Properties

    let unitsNotation: UnitsNotation

    // MARK: - Public Interface
```

```
10
11      var text: String {
12          switch unitsNotation {
13          case .imperial: return "Imperial"
14          case .metric: return "Metric"
15          }
16      }
17
18      var accessoryType: UITableViewCellAccessoryType {
19          if UserDefaults.unitsNotation() == unitsNotation {
20              return .checkmark
21          } else {
22              return .none
23          }
24      }
25
26  }
```

The implementation of the `SettingsViewTemperatureViewModel` struct looks very similar as you can see below.

SettingsViewTemperatureViewModel.swift

```
 1  import UIKit
 2
 3  struct SettingsViewTemperatureViewModel {
 4
 5      // MARK: - Properties
 6
 7      let temperatureNotation: TemperatureNotation
 8
 9      // MARK: - Public Interface
10
11      var text: String {
12          switch temperatureNotation {
13          case .fahrenheit: return "Fahrenheit"
14          case .celsius: return "Celsius"
15          }
16      }
17
18      var accessoryType: UITableViewCellAccessoryType {
19          if UserDefaults.temperatureNotation() == temperatureNotation\
20   {
21              return .checkmark
22          } else {
23              return .none
24          }
25      }
26
27  }
```

The implementation of the `tableView(_:cellForRowAt:)` method isn't difficult to update either. In the units section, we create an instance of the `UnitsNotation` enum, use it to instantiate the view model, and configure the table view cell.

SettingsViewController.swift

```
case .units:
    guard let unitsNotation = UnitsNotation(rawValue: indexPath.row)\
 else { fatalError("Unexpected Index Path") }

    // Initialize View Model
    let viewModel = SettingsViewUnitsViewModel(unitsNotation: unitsN\
otation)

    // Configure Cell
    cell.mainLabel.text = viewModel.text
    cell.accessoryType = viewModel.accessoryType
```

In the temperature section, we create an instance of the `TemperatureNotation` enum, use it to instantiate the view model, and configure the table view cell.

SettingsViewController.swift

```
case .temperature:
    guard let temperatureNotation = TemperatureNotation(rawValue: in\
dexPath.row) else { fatalError("Unexpected Index Path") }

    // Initialize View Model
    let viewModel = SettingsViewTemperatureViewModel(temperatureNota\
tion: temperatureNotation)

    // Configure Cell
    cell.mainLabel.text = viewModel.text
    cell.accessoryType = viewModel.accessoryType
```

Even though we simplified the `tableView(_:cellForRowAt:)` method of the `SettingsViewController` class, there's room for improvement. You may have noticed that we have duplications in the `tableView(_:cellForRowAt:)` method. We resolve these issues later in the book using **protocol-oriented programming**.

I'm sure you agree that the view models we created are lightweight objects. Creating and discarding them doesn't impact performance because they're inexpensive to create.

Have you noticed that the view models are short-lived objects? The settings view controller doesn't keep a reference to the view models. They're created in the `tableView(_:cellForRowAt:)` method of the settings view controller and discarded soon after the method returns a

table view cell. This is fine, though. It’s a lightweight flavor of the Model-View-ViewModel pattern.

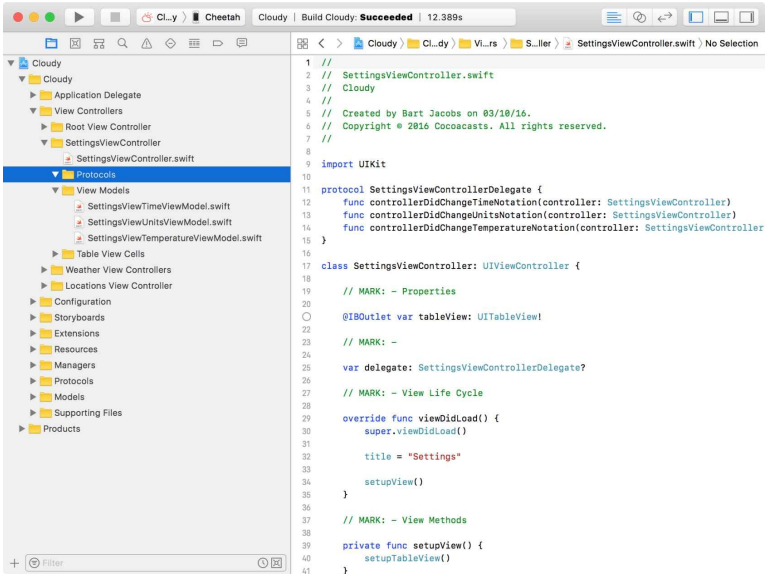
10 Adding Protocols to the Mix

After we implemented the Model-View-ViewModel pattern in the settings view controller, we noticed we were repeating ourselves in the `tableView(_:cellForRowAt:)` method. We can solve this problem with a pinch of **protocol-oriented programming**.

The idea is simple. The view models for each of the sections of the table view are very similar and the computed properties the view controller accesses have the same name. This means we can create a protocol with an interface identical to those of the view models. Let me show you what I mean.

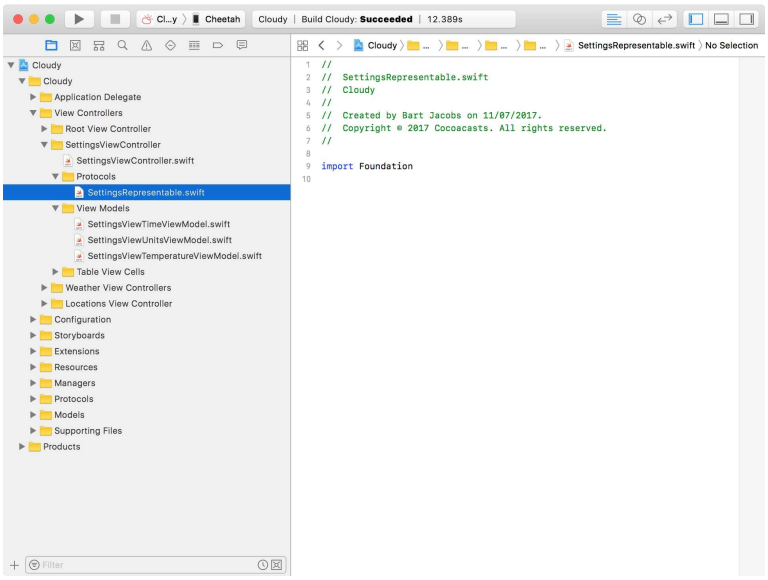
Creating the Protocol

Create a new group in the **Settings View Controller** group and name it **Protocols**. I prefer to keep protocols that are very specific close to where they're used. But that's just a personal preference.



Creating the Protocols Group

Create a Swift file and name it **SettingsRepresentable.swift**.



Creating SettingsRepresentable.swift

Replace the import statement for **Foundation** with an import statement for **UIKit** and declare the `SettingsRepresentable` protocol.

SettingsRepresentable.swift

```
1 import UIKit
2
3 protocol SettingsRepresentable {
4
5 }
```

The protocol declares two properties, `text` of type `String` and `accessoryType` of type `UITableViewCellAccessoryType`.

SettingsRepresentable.swift

```
1 import UIKit
2
3 protocol SettingsRepresentable {
4
5     var text: String { get }
6     var accessoryType: UITableViewCellAccessoryType { get }
7
8 }
```

Conforming to the Protocol

The next step is surprisingly easy. Because the three view models implicitly conform to the `SettingsRepresentable` protocol, we only need to

make their conformance to the protocol explicit. We use an extension to accomplish this.

SettingsViewTimeViewModel.swift

```
1  import UIKit
2
3  struct SettingsViewTimeViewModel {
4
5      ...
6
7  }
8
9  extension SettingsViewTimeViewModel: SettingsRepresentable {
10
11 }
```

SettingsViewUnitsViewModel.swift

```
1  import UIKit
2
3  struct SettingsViewUnitsViewModel {
4
5      ...
6
7  }
8
9  extension SettingsViewUnitsViewModel: SettingsRepresentable {
10
11 }
```

SettingsViewTemperatureViewModel.swift

```
1  import UIKit
2
3  struct SettingsViewTemperatureViewModel {
4
5      ...
6
7  }
8
9  extension SettingsViewTemperatureViewModel: SettingsRepresentable {
10
11 }
```

Are you wondering why the extension is empty? We only use the extension to make the conformance of the view models to the `SettingsRepresentable` protocol explicit. There's no need to implement the `text` and `accessoryType` properties of the protocol since the view models already implement these properties. This is important to understand.

Refactoring the Settings View Controller

It's time to update the `SettingsViewController` class. Open **SettingsViewController.swift** and navigate to the `tableView(_:cellForRowAt:)` method. Before entering the `switch` statement, we declare a variable, `viewModel`, of type `SettingsRepresentable?`.

SettingsViewController.swift

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: Ind\
exPath) -> UITableViewCell {
    guard let section = Section(rawValue: indexPath.section) else { \
fatalError("Unexpected Section") }
    guard let cell = tableView.dequeueReusableCell(withIdentifier: S\
ettingsTableViewCell.reuseIdentifier, for: indexPath) as? SettingsTa\
bleViewCell else { fatalError("Unexpected Table View Cell") }

    // Helpers
    var viewModel: SettingsRepresentable?

    switch section {
        ...
    }

    return cell
}
```

We set the value of this variable in the `switch` statement. We can remove the configuration of the table view cell from the `switch` statement and move it to the bottom of the implementation of the `tableView(_:cellForRowAt:)` method. That's a first step in the right direction.

SettingsViewController.swift

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: Ind\
exPath) -> UITableViewCell {
    guard let section = Section(rawValue: indexPath.section) else { \
fatalError("Unexpected Section") }
    guard let cell = tableView.dequeueReusableCell(withIdentifier: S\
ettingsTableViewCell.reuseIdentifier, for: indexPath) as? SettingsTa\
bleViewCell else { fatalError("Unexpected Table View Cell") }

    // Helpers
    var viewModel: SettingsRepresentable?

    switch section {
    case .time:
        guard let timeNotation = TimeNotation(rawValue: indexPath.ro\
w) else { fatalError("Unexpected Index Path") }
        viewModel = SettingsViewTimeViewModel(timeNotation: timeNota\
tion)
```

```
    case .units:
        guard let unitsNotation = UnitsNotation(rawValue: indexPath.\
row) else { fatalError("Unexpected Index Path") }
        viewModel = SettingsViewUnitsViewModel(unitsNotation: unitsN\
otation)
    case .temperature:
        guard let temperatureNotation = TemperatureNotation(rawValue\
: indexPath.row) else { fatalError("Unexpected Index Path") }
        viewModel = SettingsViewTemperatureViewModel(temperatureNota\
tion: temperatureNotation)
    }

    if let viewModel = viewModel {
        // Configure Cell
        cell.mainLabel.text = viewModel.text
        cell.accessoryType = viewModel.accessoryType
    }

    return cell
}
```

Protocols work very well with the Model-View-ViewModel pattern. In the next chapter, we take it one step further.

11 Making Table View Cells Autoconfigurable

I don't know if autoconfigurable can be found in the dictionary, but I'm going to use it anyway because it best describes what we're going to do in this chapter. In the previous chapter, we refactored the `SettingsViewController` class. In the `tableView(_:cellForRowAt:)` method, we create an instance of type `SettingsRepresentable?`. The view controller manually configures each table view cell, using a view model. But why can't the table view cell take care of its own configuration?

All the table view cell needs to do is ask the view model for the values it needs and populate itself with data. This is very different from handing a model to a view, which certainly isn't what we intend to do. The table view cell won't know about the model, it will simply use the interface of the view model we give it.

Updating the Settings Table View Cell

To make this work, we need to make some changes to the `SettingsTableViewCell` class. Open **SettingsTableViewCell.swift** and define a method named `configure(withViewModel:)`. The method accepts one argument of type `SettingsRepresentable`. In the body of the method, we set the `text` property of the `mainLabel` outlet and we update the `accessoryType` property of the table view cell.

SettingsTableViewCell.swift

```
1  import UIKit
2
3  class SettingsTableViewCell: UITableViewCell {
4
5      // MARK: - Type Properties
6
7      static let reuseIdentifier = "SettingsCell"
8
9      // MARK: - Properties
10
11     @IBOutlet var mainLabel: UILabel!
12
13     ...
14
```

```
15      // MARK: - Configuration
16
17      func configure(withViewModel viewModel: SettingsRepresentable) {
18          mainLabel.text = viewModel.text
19          accessoryType = viewModel.accessoryType
20      }
21
22  }
```

Updating the Settings View Controller

All that's left for us to do is updating the `SettingsViewController` class. Open **SettingsViewController.swift** and navigate to the `tableView(_:cellForRowAt:)` method. The view controller no longer needs to configure the table view cell. Instead, we invoke the `configure(withViewModel:)` method of the table view cell and pass in the view model. I'm sure you agree that this looks quite nice.

SettingsViewController.swift

```
 1  func tableView(_ tableView: UITableView, cellForRowAt indexPath: Ind\
 2  exPath) -> UITableViewCell {
 3      ...
 4
 5      if let viewModel = viewModel {
 6          cell.configure(withViewModel: viewModel)
 7      }
 8
 9      return cell
10  }
```

Notice that the view controller is still involved, but its role is very limited. By implementing the `configure(withViewModel:)` method and handing a view model to the table view cell, we've slightly deviated from the Model-View-ViewModel pattern we discussed in the introduction of this book.

But that's fine. Views need to be dumb. They shouldn't know what they're displaying. That still applies to the `SettingsTableViewCell` class. The table view cell only uses the interface of the view model to configure itself.

Protocol-Oriented Programming

I want to highlight the role of the `SettingsRepresentable` protocol in the story. The `SettingsRepresentable` protocol serves two purposes. The most obvious task of the `SettingsRepresentable` protocol is defining an

interface. Because the view models we defined earlier conform to this protocol, the `SettingsTableViewCell` class only needs to have the ability to handle an object of type `SettingsRepresentable`.

But the `SettingsRepresentable` protocol performs a more important but less obvious task. It adds a layer of abstraction. The `SettingsRepresentable` protocol ensures that the `SettingsTableViewCell` class doesn't need to know about the view models we defined (`SettingsViewTimeViewModel`, `SettingsViewUnitsViewModel`, and `SettingsViewTemperatureViewModel`). That's an important advantage and that's the beauty and elegance of protocol-oriented programming.

Conclusion

The settings view controller is a simple view controller, but I hope you can see the potential of the Model-View-ViewModel pattern in simplifying view controllers. The settings view controller is very focused. All it does is manage its view and subviews and handle user interaction. That's the primary role of every view controller, and this applies to MVVM as well as MVC.

In the next chapter, we apply what we've learned in the past two chapter to the week view controller.

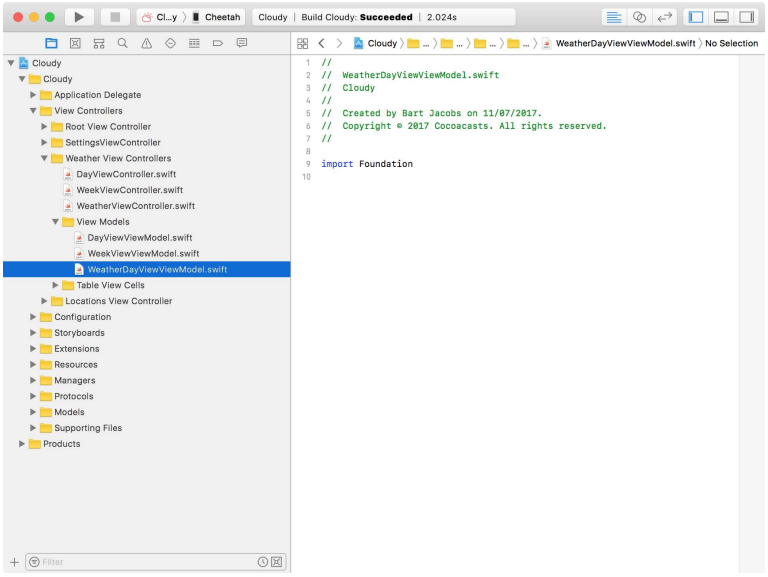
12 Supercharging MVVM With Protocols

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

```
import UIKit

struct WeekViewViewModel {

    // MARK: - Properties

    let weatherData: [WeatherDayData]

    // MARK: -

    var numberOfSections: Int {
        return 1
    }

    var numberOfDays: Int {
        return weatherData.count
    }

}
```

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

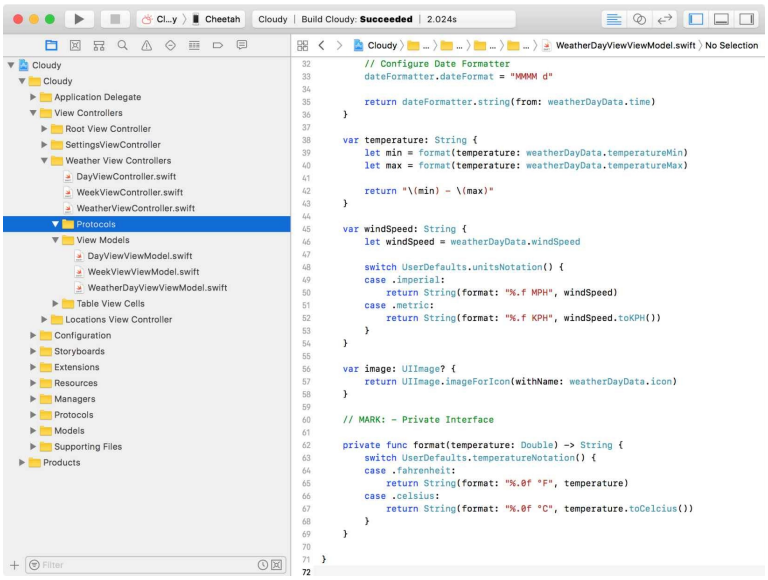
```
func viewModel(for index: Int) -> WeatherDayViewViewModel {
    return WeatherDayViewViewModel(weatherDayData: weatherData[index\
])
}
```

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

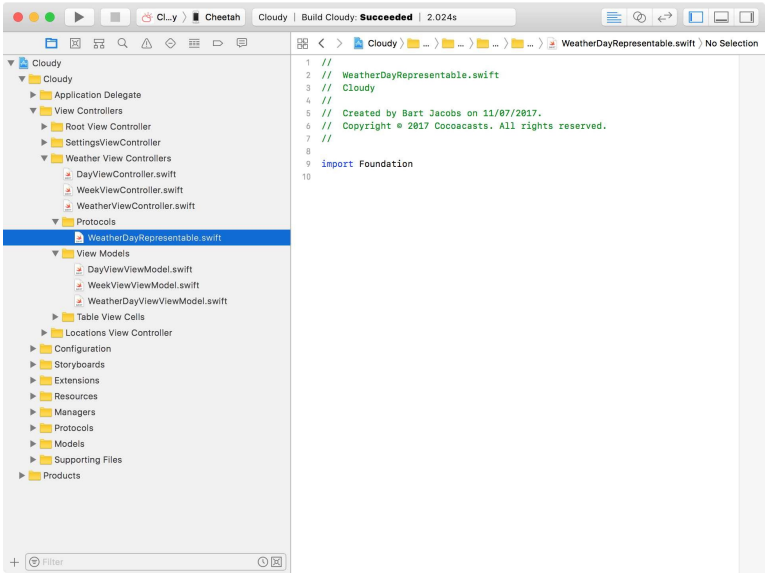
We could pass the `WeatherDayViewViewModel` instance directly to the table view cell, but, as I explained earlier, I prefer to use a protocol to define the interface the table view cell expects. Remember that this adds a layer of abstraction between the view model layer and the view layer.

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

```
func configure(withViewModel viewModel: WeatherDayRepresentable) {
    dayLabel.text = viewModel.day
    dateLabel.text = viewModel.date
    iconImageView.image = viewModel.image
    windSpeedLabel.text = viewModel.windSpeed
    temperatureLabel.text = viewModel.temperature
}
```

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

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: Ind\
exPath) -> UITableViewCell {
    guard let cell = tableView.dequeueReusableCell(withIdentifier: W\
eatherDayTableViewCell.reuseIdentifier, for: indexPath) as? WeatherD\
ayTableViewCell else { fatalError("Unexpected Table View Cell") }

    if let weatherDayRepresentable = viewModel?.viewModel(for: index\
Path.row) {
        cell.configure(withViewModel: weatherDayRepresentable)
    }

    return cell
}
```

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.

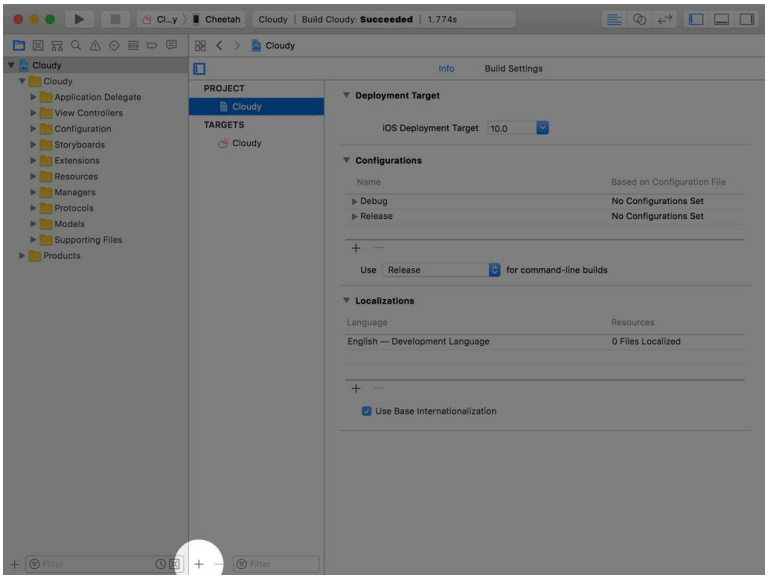
I hope that the past few chapters have convinced you of the benefits of the Model-View-ViewModel pattern. Not only does it result in a clear separation of responsibilities, the testability of the project has improved substantially. And that's something we look at in the next chapters.

13 Ready, Set, Test

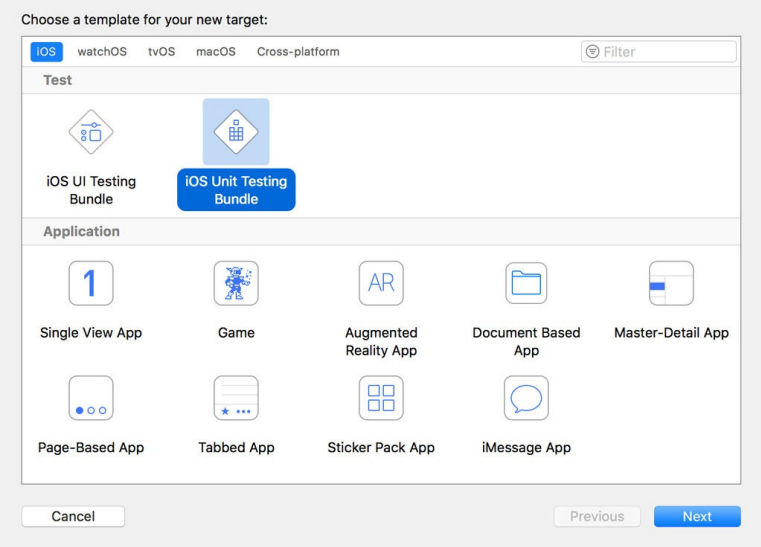
Because we moved a fair bit of logic from the view controllers of the project into the view models, we gained an important advantage, **improved testability**. As I mentioned earlier, unit testing view controllers is known to be difficult. View models, however, are easy to test. And that's what I'll 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**.

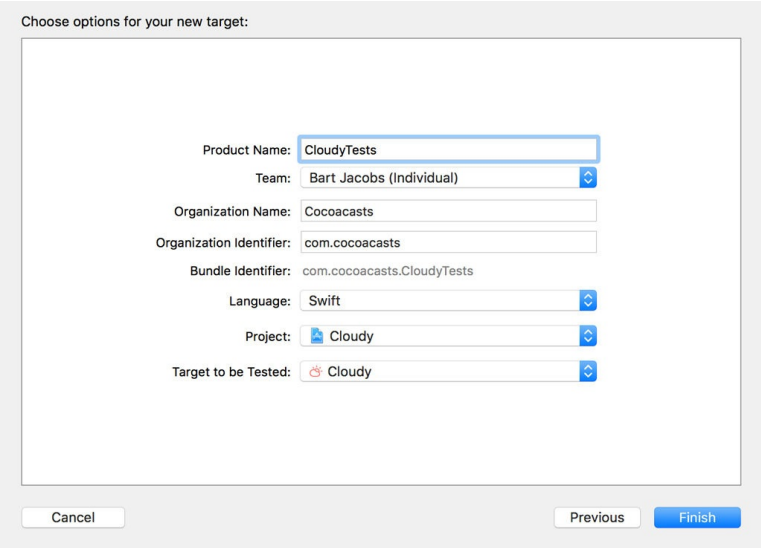


Adding a Target



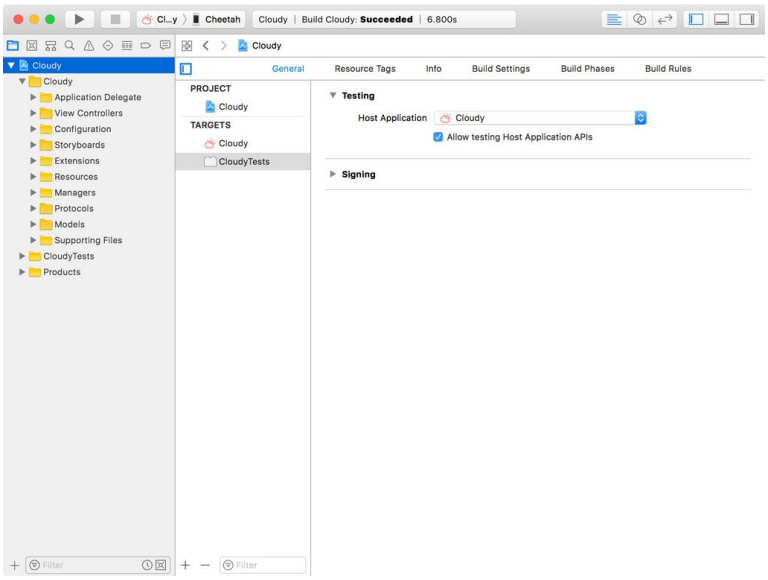
Choosing the iOS Unit Testing Bundle Template

The defaults are just fine. Make sure **Language** is set to **Swift** and **Target to be Tested** is set to **Cloudy**.



Configuring the Target

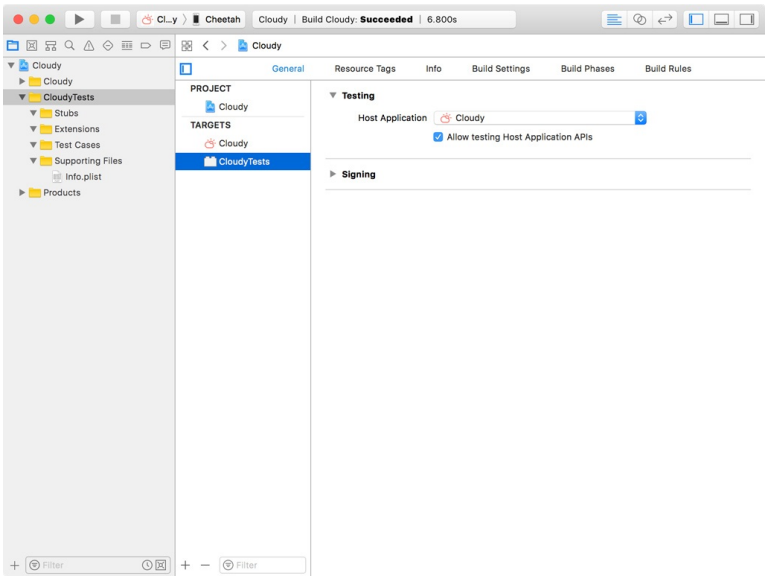
The **Cloudy** project should now have two targets, **Cloudy** and **CloudyTests**.



The Cloudy project should now have two targets.

Organizing the Unit Test Target

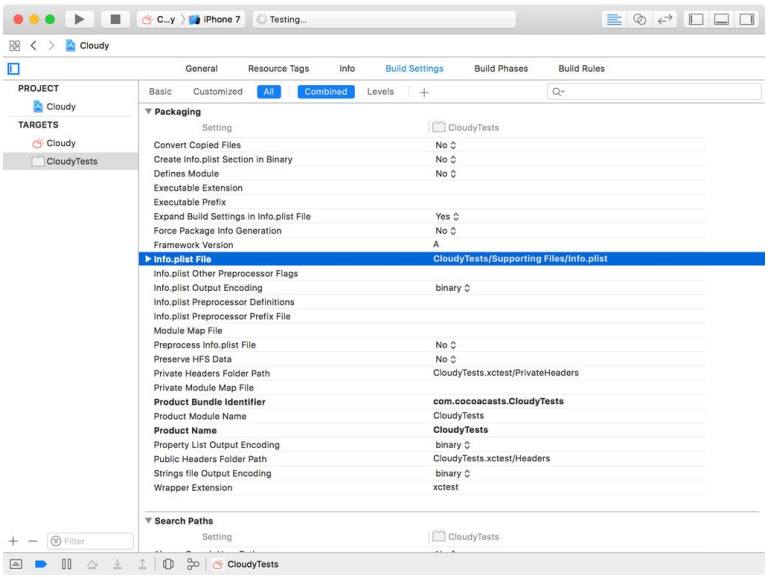
I usually create several groups in the unit testing bundle to keep files and folders organized. I create a group named **Supporting Files** for the **Info.plist** file, a group for stubs, a group for extensions, and a group for the test cases. This is what you should end up with. I also removed the test 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**.

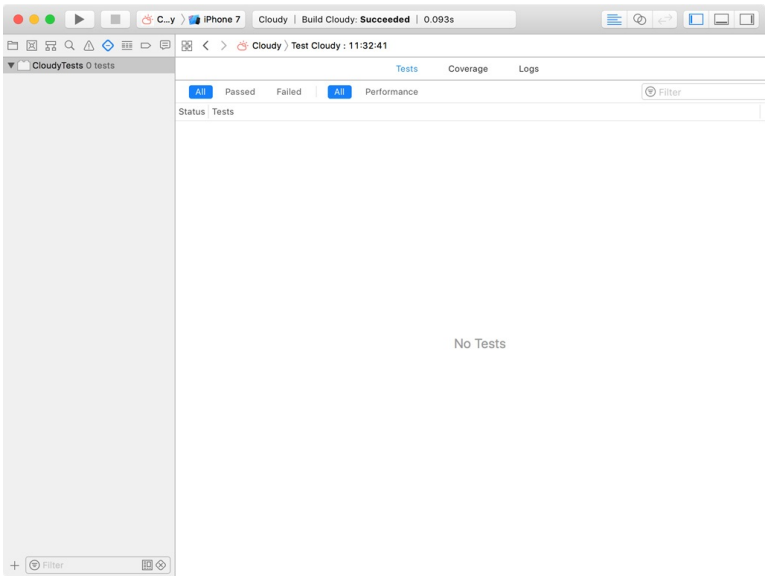
We also need to update the path to the **Info.plist** file in the build settings of the **CloudyTests** target. Choose the **CloudyTests** target from the list of targets, select **Build Settings** at the top, and search for the **Info.plist File** build setting in the **Packaging** section. Change the 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.



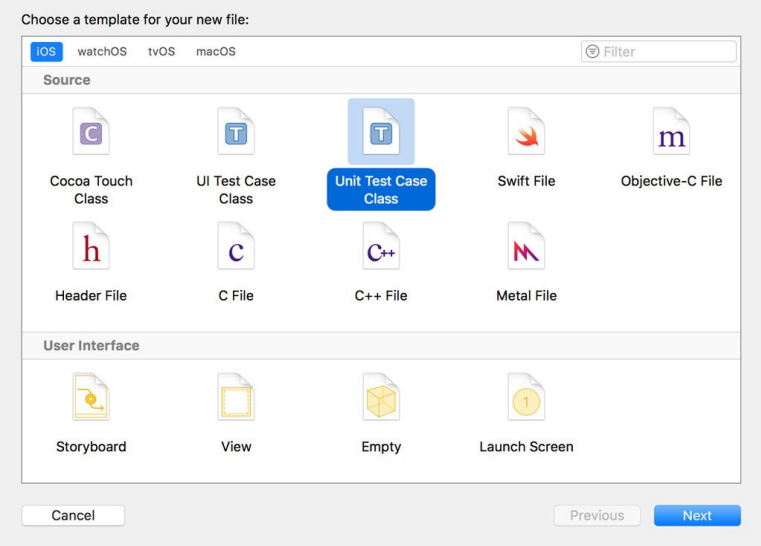
Running the Test Suite

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

14 Testing Your First View Model

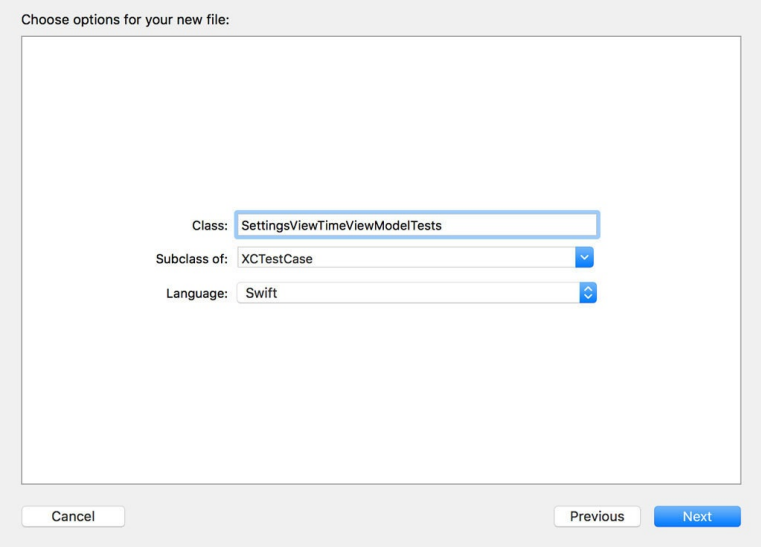
Creating a Test Case

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

```
import XCTest
@testable import Cloudy

class SettingsViewTimeViewModelTests: XCTestCase {

    // MARK: - Set Up & Tear Down

    override func setUp() {
        super.setUp()
    }

    override func tearDown() {
        super.tearDown()
    }

}
```

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

```
var text: String {
    switch timeNotation {
    case .twelveHour: return "12 Hour"
    case .twentyFourHour: return "24 Hour"
    }
}
```

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

```
// MARK: - Tests for Text

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.

```
11
class SettingsViewTimeViewModelTests: XCTestCase {
13
```

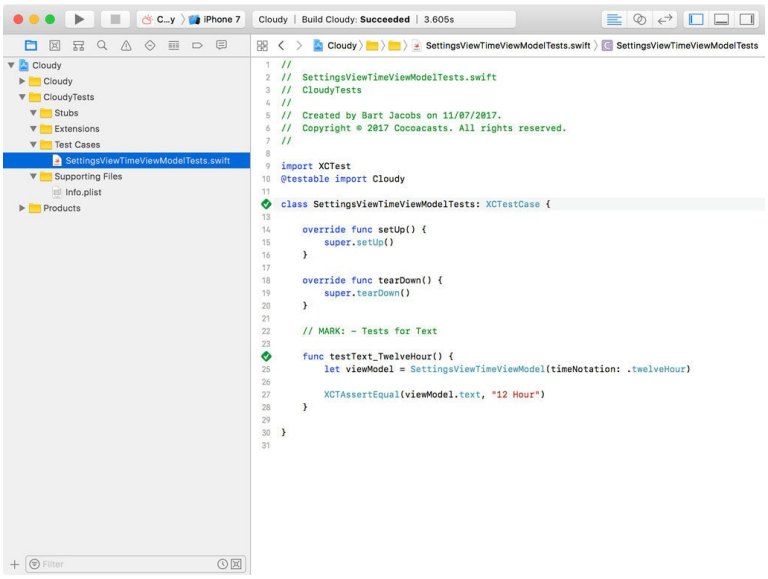
Running Unit Tests of a XCTestCase Subclass

To run one unit test, we click the diamond next to the test we're interested in.

```
23
    func testText_TwelveHour() {
25      let viewModel = SettingsViewTimeViewModel(timeNotation: .twelv
26
```

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

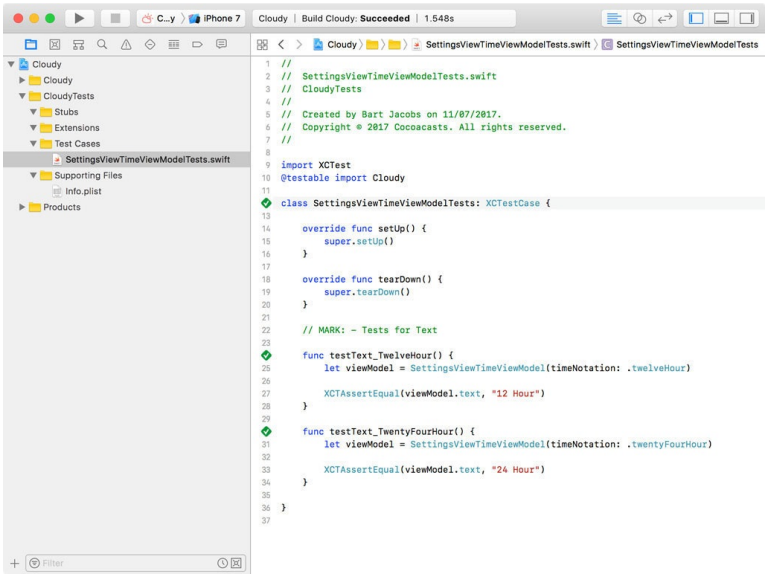
The unit test passed.

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

```
import XCTest
@testable import Cloudy

class SettingsViewTimeViewModelTests: XCTestCase {

    // MARK: - Set Up & Tear Down

    override func setUp() {
        super.setUp()
    }

    override func tearDown() {
        super.tearDown()
    }

    // MARK: - Tests for Text

    func testText_TwelveHour() {
     ...
    }

    func testText_TwentyFourHour() {
     ...
    }

}
```

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

```
var accessoryType: UITableViewCellAccessoryType {
    if UserDefaults.timeNotation() == timeNotation {
        return .checkmark
    } else {
        return .none
    }
}
```

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

```
// MARK: - Tests for Accessory Type

func testAccessoryType_TwelveHour_TwelveHour() {

}
```

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

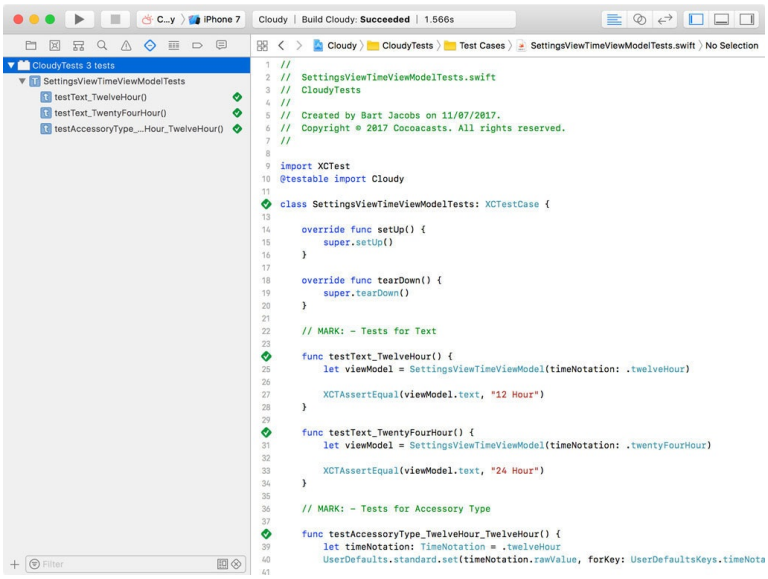
```
// MARK: - Tests for Accessory Type

func testAccessoryType_TwelveHour_TwelveHour() {
    let timeNotation: TimeNotation = .twelveHour
    UserDefaults.standard.set(timeNotation.rawValue, forKey: UserDef\
aultsKeys.timeNotation)

    let viewModel = SettingsViewTimeViewModel(timeNotation: .twelveH\
our)
}
```

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

```
// MARK: - Tests for Accessory Type

func testAccessoryType_TwelveHour_TwelveHour() {
    let timeNotation: TimeNotation = .twelveHour
    UserDefaults.standard.set(timeNotation.rawValue, forKey: UserDef\
aultsKeys.timeNotation)

    let viewModel = SettingsViewTimeViewModel(timeNotation: .twelveH\
our)

    XCTAssertEqual(viewModel.accessoryType, UITableViewCellAccessory\
Type.checkmark)
}
```

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.



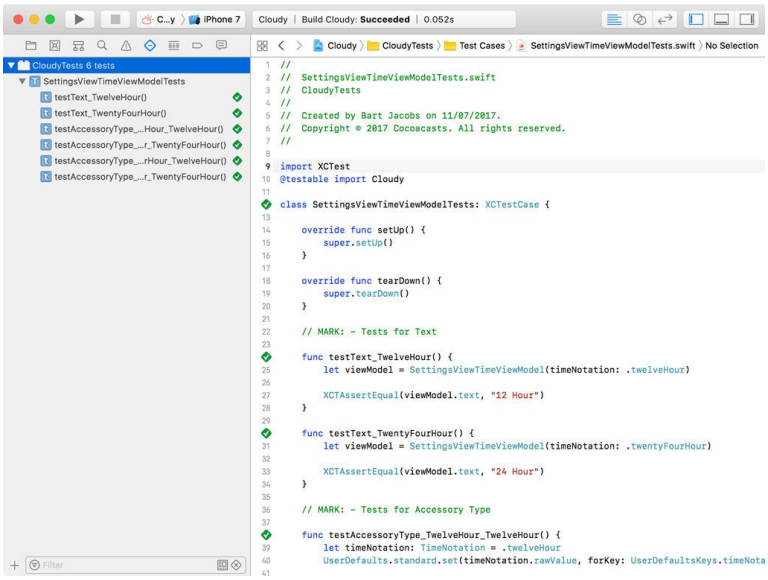
Bringing Up the Test Navigator

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

```
import XCTest
@testable import Cloudy

class SettingsViewTimeViewModelTests: XCTestCase {

    ...
```

```
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

```
override func tearDown() {
    super.tearDown()

    // Reset User Defaults
    UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.time\
Notation)
}
```

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

```
import XCTest
@testable import Cloudy

class SettingsViewUnitsViewModelTests: XCTestCase {

    // MARK: - Set Up & Tear Down

    override func setUp() {
        super.setUp()
    }

    override func tearDown() {
        super.tearDown()

        // Reset User Defaults
        UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.\
unitsNotation)
    }

    // MARK: - Tests for Text

    func testText_Imperial() {
        let viewModel = SettingsViewUnitsViewModel(unitsNotation: .i\
mperial)

        XCTAssertEqual(viewModel.text, "Imperial")
    }

    func testText_Metric() {
        let viewModel = SettingsViewUnitsViewModel(unitsNotation: .m\
etric)

        XCTAssertEqual(viewModel.text, "Metric")
    }

    // MARK: - Tests for Accessory Type

    func testAccessoryType_Imperial_Imperial() {
        let unitsNotation: UnitsNotation = .imperial
        UserDefaults.standard.set(unitsNotation.rawValue, forKey: Us\
erDefaultsKeys.unitsNotation)

        let viewModel = SettingsViewUnitsViewModel(unitsNotation: .i\
```

```
mperial)

        XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\
soryType.checkmark)
    }


    func testAccessoryType_Imperial_Metric() {
        let unitsNotation: UnitsNotation = .imperial
        UserDefaults.standard.set(unitsNotation.rawValue, forKey: Us\
erDefaultsKeys.unitsNotation)

        let viewModel = SettingsViewUnitsViewModel(unitsNotation: .m\
etric)

        XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\
soryType.none)
    }

    func testAccessoryType_Metric_Imperial() {
        let unitsNotation: UnitsNotation = .metric
        UserDefaults.standard.set(unitsNotation.rawValue, forKey: Us\
erDefaultsKeys.unitsNotation)

        let viewModel = SettingsViewUnitsViewModel(unitsNotation: .i\
mperial)

        XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\
soryType.none)
    }


    func testAccessoryType_Metric_Metric() {
        let unitsNotation: UnitsNotation = .metric
        UserDefaults.standard.set(unitsNotation.rawValue, forKey: Us\
erDefaultsKeys.unitsNotation)

        let viewModel = SettingsViewUnitsViewModel(unitsNotation: .m\
etric)

        XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\
soryType.checkmark)
    }

}
```

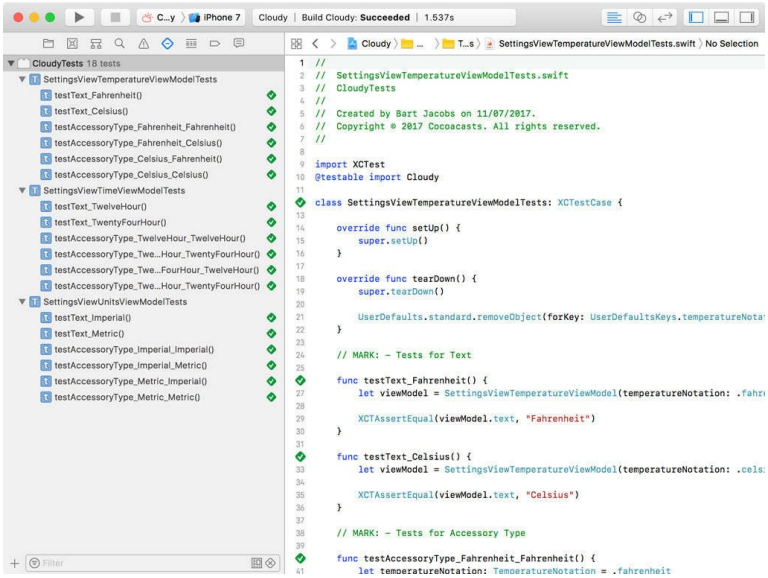
SettingsViewTemperatureViewModelTests.swift

```swift
import XCTest
@testable import Cloudy

class SettingsViewTemperatureViewModelTests: XCTestCase {

    // MARK: - Set Up & Tear Down

    override func setUp() {
        super.setUp()
    }

    override func tearDown() {
        super.tearDown()

```

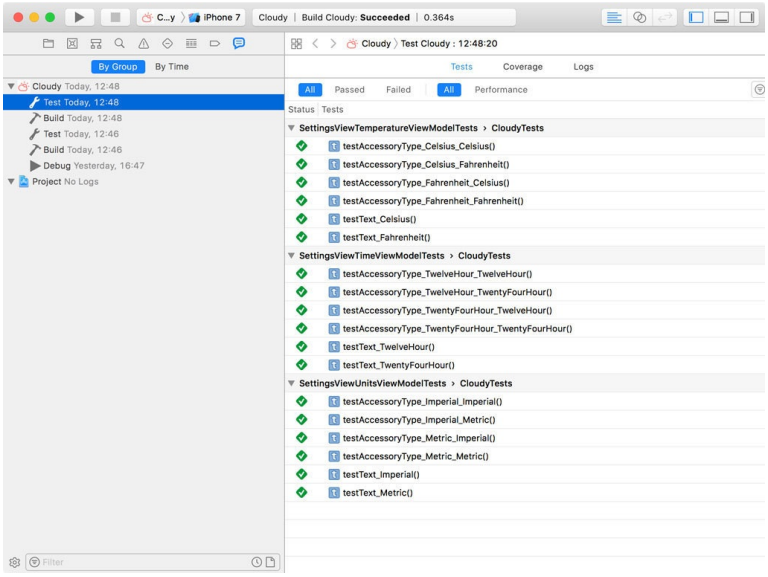
```
        // Reset User Defaults
        UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.\
temperatureNotation)
    }

    // MARK: - Tests for Text

    func testText_Fahrenheit() {
        let viewModel = SettingsViewTemperatureViewModel(temperature\
Notation: .fahrenheit)

        XCTAssertEqual(viewModel.text, "Fahrenheit")
    }

    func testText_Celsius() {
        let viewModel = SettingsViewTemperatureViewModel(temperature\
Notation: .celsius)

        XCTAssertEqual(viewModel.text, "Celsius")
    }

    // MARK: - Tests for Accessory Type

    func testAccessoryType_Fahrenheit_Fahrenheit() {
        let temperatureNotation: TemperatureNotation = .fahrenheit
        UserDefaults.standard.set(temperatureNotation.rawValue, forK\
ey: UserDefaultsKeys.temperatureNotation)

        let viewModel = SettingsViewTemperatureViewModel(temperature\
Notation: .fahrenheit)

        XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\
soryType.checkmark)
    }

    func testAccessoryType_Fahrenheit_Celsius() {
        let temperatureNotation: TemperatureNotation = .fahrenheit
        UserDefaults.standard.set(temperatureNotation.rawValue, forK\
ey: UserDefaultsKeys.temperatureNotation)

        let viewModel = SettingsViewTemperatureViewModel(temperature\
Notation: .celsius)

        XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\
soryType.none)
    }

    func testAccessoryType_Celsius_Fahrenheit() {
        let temperatureNotation: TemperatureNotation = .celsius
        UserDefaults.standard.set(temperatureNotation.rawValue, forK\
ey: UserDefaultsKeys.temperatureNotation)

        let viewModel = SettingsViewTemperatureViewModel(temperature\
Notation: .fahrenheit)

        XCTAssertEqual(viewModel.accessoryType, UITableViewCellAcces\
soryType.none)
    }

    func testAccessoryType_Celsius_Celsius() {
        let temperatureNotation: TemperatureNotation = .celsius
        UserDefaults.standard.set(temperatureNotation.rawValue, forK\
ey: UserDefaultsKeys.temperatureNotation)

```

```
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

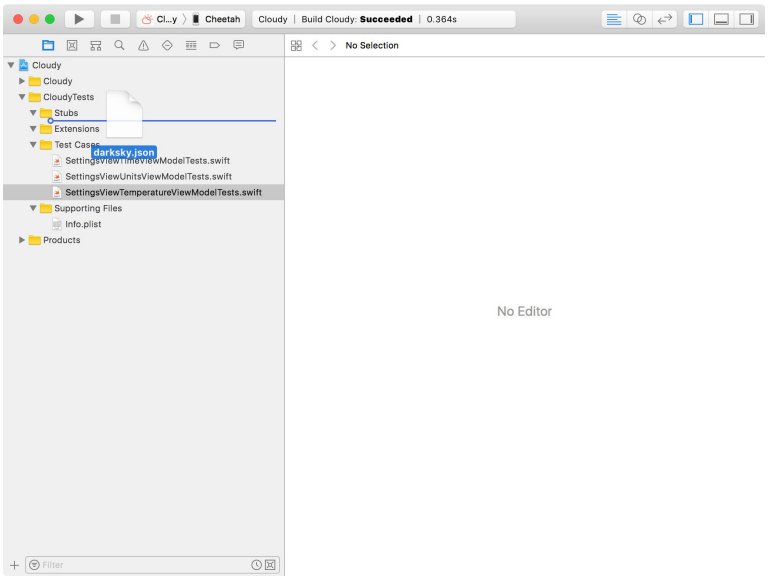
Testing the `DayViewViewModel` struct isn't very different from testing the view models of the `SettingsViewController` class. The only tricky aspect is instantiating a `DayViewViewModel` instance in a unit test.

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.

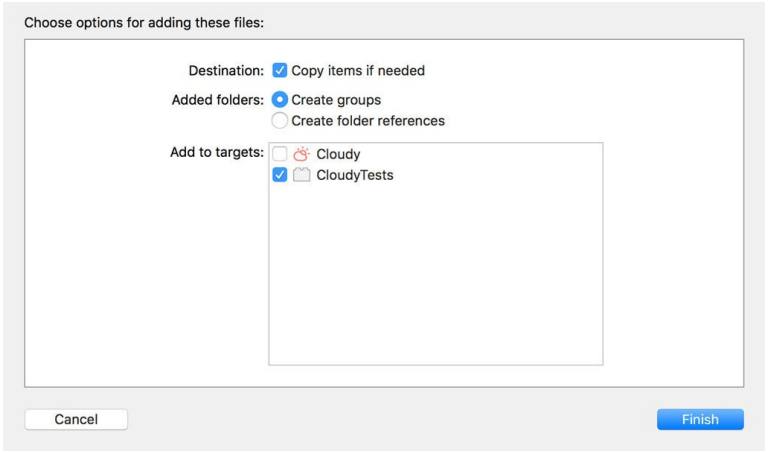
Adding Stub Data

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.

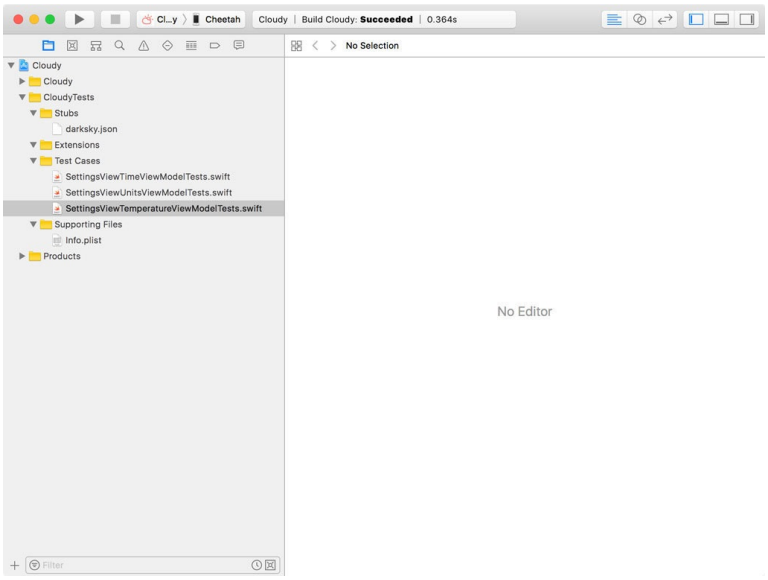


Adding Stub Data

Make sure that **Copy items if needed** is checked and that the file is only added to the **CloudyTests** target.



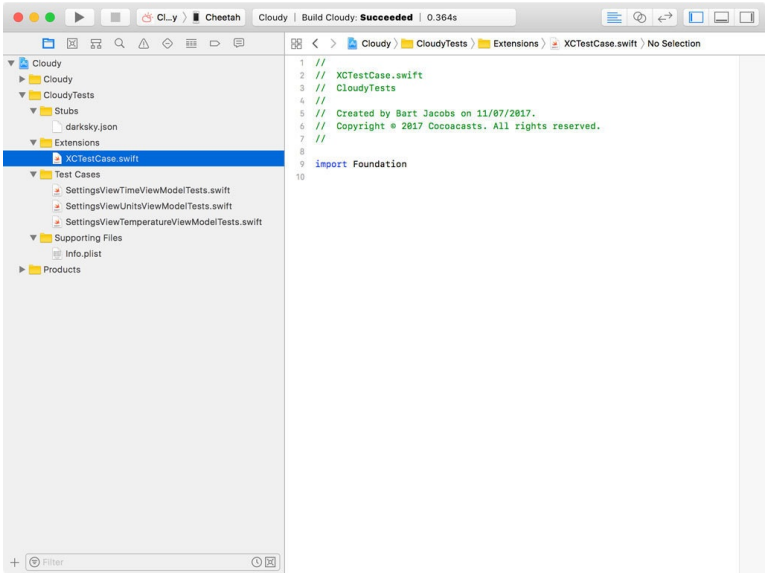
Adding Stub Data to CloudyTests Target



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.

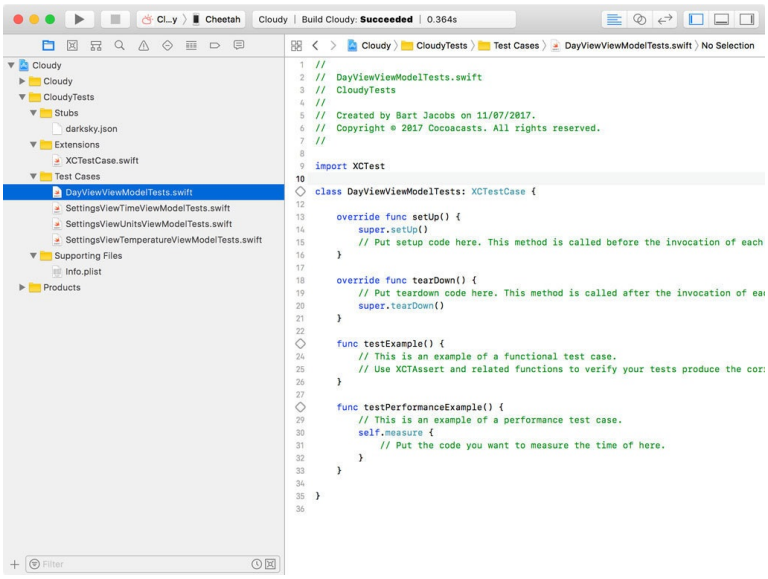
XCTestCase.swift

```
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

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

```
import XCTest
@testable import Cloudy

class DayViewViewModelTests: XCTestCase {

    // MARK: - Set Up & Tear Down

    override func setUp() {
        super.setUp()
    }

    override func tearDown() {
        super.tearDown()
    }

}
```

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.

DayViewViewModelTests.swift

```
import XCTest
@testable import Cloudy

class DayViewViewModelTests: XCTestCase {

    // MARK: - Properties

    var viewModel: DayViewViewModel!

    // MARK: - Set Up & Tear Down

    override func setUp() {
        super.setUp()
    }

    override func tearDown() {
        super.tearDown()
    }

}
```

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

```
override func setUp() {
    super.setUp()

    // Load Stub
    let data = loadStubFromBundle(withName: "darksky", extension: "j\
son")
    let weatherData: WeatherData = try! JSONDecoder.decode(data: dat\
a)

    // 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

```
func testTime_TwelveHour() {
    let timeNotation: TimeNotation = .twelveHour
    UserDefaults.standard.set(timeNotation.rawValue, forKey: UserDef\
aultsKeys.timeNotation)

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

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

```
func testTime_TwentyFourHour() {
    let timeNotation: TimeNotation = .twentyFourHour
    UserDefaults.standard.set(timeNotation.rawValue, forKey: UserDef\
aultsKeys.timeNotation)

    XCTAssertEqual(viewModel.time, "13:57")
}
```

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

```
// MARK: - Tests for Summary

func testSummary() {
    XCTAssertEqual(viewModel.summary, "Clear")
}

// MARK: - Tests for Temperature

func testTemperature_Fahrenheit() {
    let temperatureNotation: TemperatureNotation = .fahrenheit
    UserDefaults.standard.set(temperatureNotation.rawValue, forKey: \
UserDefaultsKeys.temperatureNotation)

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

func testTemperature_Celsius() {
    let temperatureNotation: TemperatureNotation = .celsius
    UserDefaults.standard.set(temperatureNotation.rawValue, forKey: \
UserDefaultsKeys.temperatureNotation)

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

// MARK: - Tests for Wind Speed

func testWindSpeed_Imperial() {
```

```
    let unitsNotation: UnitsNotation = .imperial
    UserDefaults.standard.set(unitsNotation.rawValue, forKey: UserDefaultsKeys.unitsNotation)

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

func testWindSpeed_Metric() {
    let unitsNotation: UnitsNotation = .metric
    UserDefaults.standard.set(unitsNotation.rawValue, forKey: UserDefaultsKeys.unitsNotation)

    print(viewModel.windSpeed)

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

// MARK: - Tests for Image

func testImage() {
    let viewModelImage = viewModel.image
    let imageDataViewModel = UIImagePNGRepresentation(viewModelImage!)!
    let imageDataReference = UIImagePNGRepresentation(UIImage(named: "clear-day")!)!

    XCTAssertNotNil(viewModelImage)
    XCTAssertEqual(viewModelImage!.size.width, 236.0)
    XCTAssertEqual(viewModelImage!.size.height, 236.0)
    XCTAssertEqual(imageDataViewModel, imageDataReference)
}
```

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

```
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

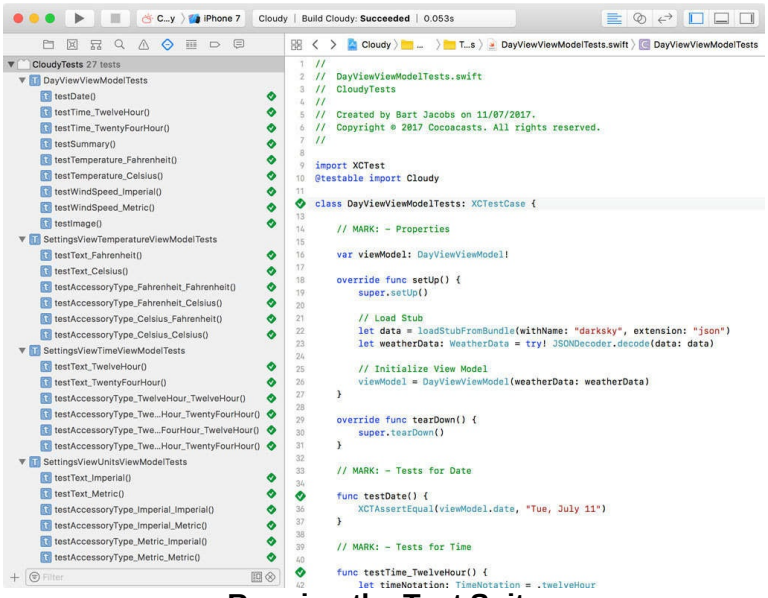
```
XCTAssertEqual(viewModelImage!.size.width, 236.0)
XCTAssertEqual(viewModelImage!.size.height, 236.0)
XCTAssertEqual(imageDataViewModel, imageDataReference)
```

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

```
override func tearDown() {
    super.tearDown()

    // Reset User Defaults
    UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.time\
Notation)
    UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.unit\
sNotation)
    UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.temp\
eratureNotation)
}
```

Press **Command + U** to run the test suite to make sure the unit tests for the `DayViewViewModel` struct pass.



Running the Test Suite

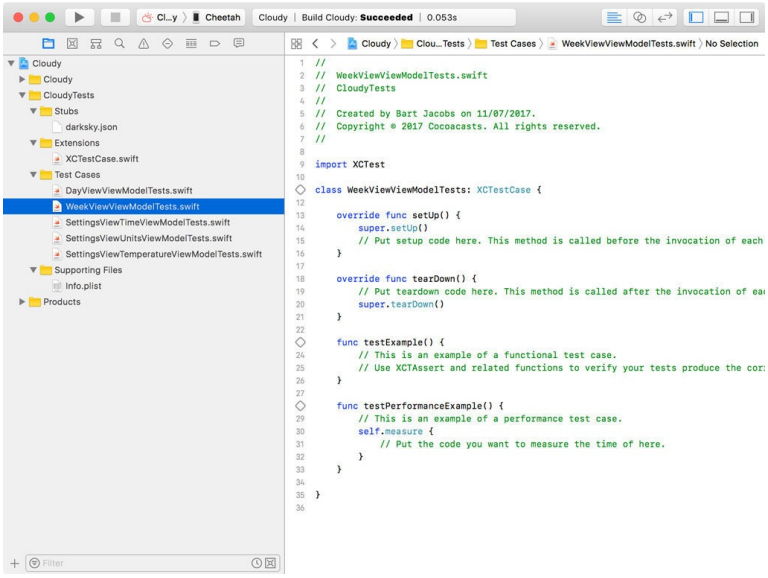
In the next chapter, we unit test the view models for the `WeekViewController` class.

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
```

```
    // MARK: - Properties

    var viewModel: WeekViewViewModel!

    // MARK: - Set Up & Tear Down

    override func setUp() {
        super.setUp()
    }

    override func tearDown() {
        super.tearDown()
    }

}
```

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

```
override func setUp() {
    super.setUp()

    // Load Stub
    let data = loadStubFromBundle(withName: "darksky", extension: "j\
son")
    let weatherData: WeatherData = try! JSONDecoder.decode(data: dat\
a)

    // Initialize View Model
    viewModel = WeekViewViewModel(weatherData: weatherData.dailyData)
}
```

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

```
// MARK: - Tests for Number of Sections

func testNumberOfSections() {
    XCTAssertEqual(viewModel.numberOfSections, 1)
}
```

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

WeekViewViewModelTests.swift

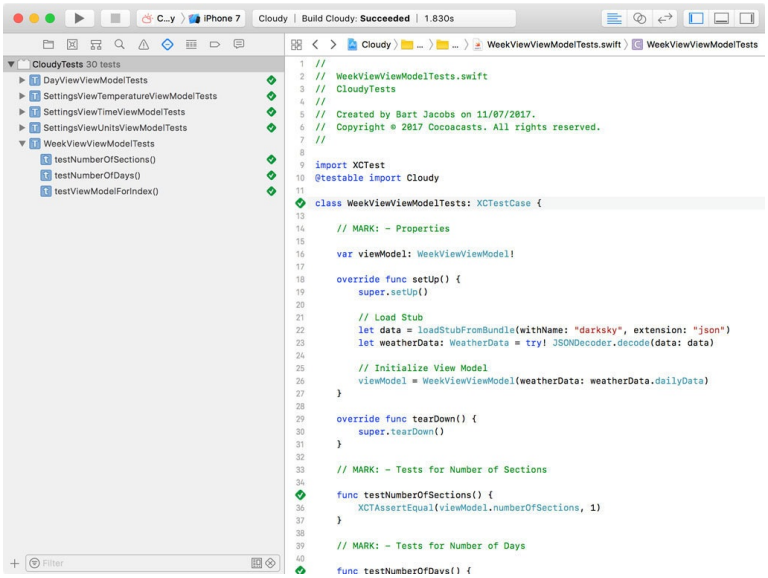
```
// MARK: - Tests for Number of Days

func testNumberOfDays() {
    XCTAssertEqual(viewModel.numberOfDays, 8)
}
```

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

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

func testViewModelForIndex() {
    let weatherDayViewViewModel = viewModel.viewModel(for: 5)

    XCTAssertEqual(weatherDayViewViewModel.day, "Saturday")
    XCTAssertEqual(weatherDayViewViewModel.date, "July 15")
}
```

These are all the unit tests we need to write for the `WeekViewViewModel` struct. Press **Command + U** to run the test suite.



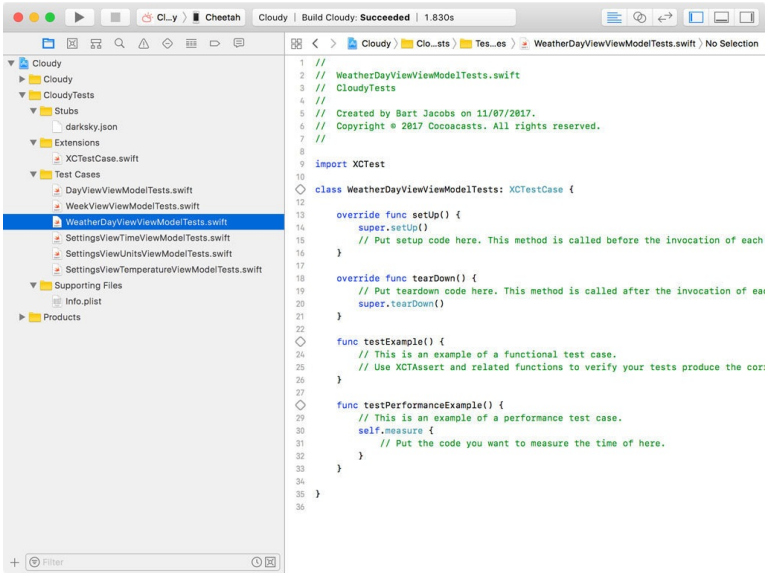
Running the Test Suite

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

```
override func setUp() {
    super.setUp()

    // Load Stub
    let data = loadStubFromBundle(withName: "darksky", extension: "j\
son")
    let weatherData: WeatherData = try! JSONDecoder.decode(data: dat\
a)

    // Initialize View Model
    viewModel = WeatherDayViewViewModel(weatherDayData: weatherData.\
dailyData[5])
}
```

The unit tests should look familiar. They're similar to the ones we wrote for the `DayViewViewModel` struct.

WeatherDayViewViewModelTests.swift

```
// MARK: - Tests for Day

func testDay() {
    XCTAssertEqual(viewModel.day, "Saturday")
}

// MARK: - Tests for Date

func testDate() {
    XCTAssertEqual(viewModel.date, "July 15")
}

// MARK: - Tests for Temperature

func testTemperature_Fahrenheit() {
    let temperatureNotation: TemperatureNotation = .fahrenheit
    UserDefaults.standard.set(temperatureNotation.rawValue, forKey: \
UserDefaultsKeys.temperatureNotation)

```

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

func testTemperature_Celsius() {
    let temperatureNotation: TemperatureNotation = .celsius
    UserDefaults.standard.set(temperatureNotation.rawValue, forKey: \
UserDefaultsKeys.temperatureNotation)

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

// MARK: - Tests for Wind Speed

func testWindSpeed_Imperial() {
    let unitsNotation: UnitsNotation = .imperial
    UserDefaults.standard.set(unitsNotation.rawValue, forKey: UserDe\
faultsKeys.unitsNotation)

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

func testWindSpeed_Metric() {
    let unitsNotation: UnitsNotation = .metric
    UserDefaults.standard.set(unitsNotation.rawValue, forKey: UserDe\
faultsKeys.unitsNotation)

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

// MARK: - Tests for Image

func testImage() {
    let viewModelImage = viewModel.image
    let imageDataViewModel = UIImagePNGRepresentation(viewModelImage\
!)!
    let imageDataReference = UIImagePNGRepresentation(UIImage(named:\
 "cloudy")!)!

    XCTAssertNotNil(viewModelImage)
    XCTAssertEqual(viewModelImage!.size.width, 236.0)
    XCTAssertEqual(viewModelImage!.size.height, 172.0)
    XCTAssertEqual(imageDataViewModel, imageDataReference)
}
```

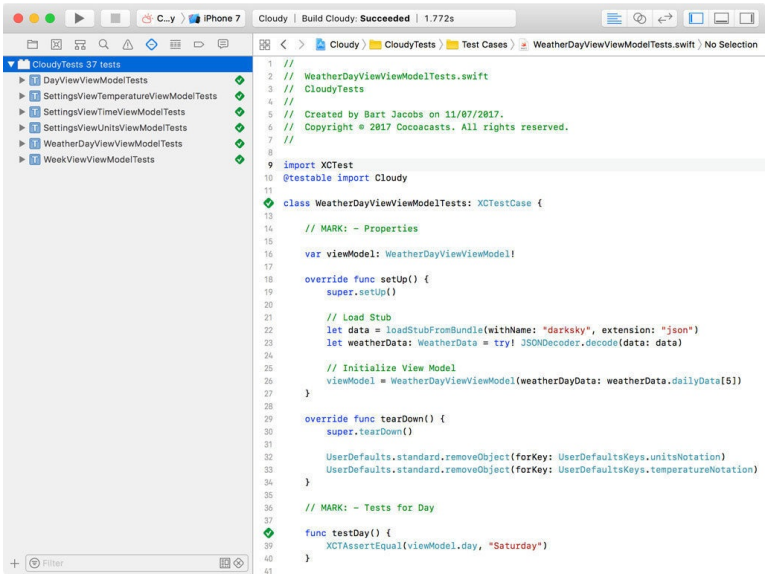
In the `tearDown()` method, we reset the state we set in the unit tests.

WeatherDayViewViewModelTests.swift

```
override func tearDown() {
    super.tearDown()

    // Reset User Defaults
    UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.unit\
sNotation)
    UserDefaults.standard.removeObject(forKey: UserDefaultsKeys.temp\
eratureNotation)
}
```

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.

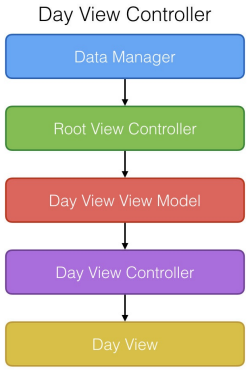


Running the Test Suite

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.

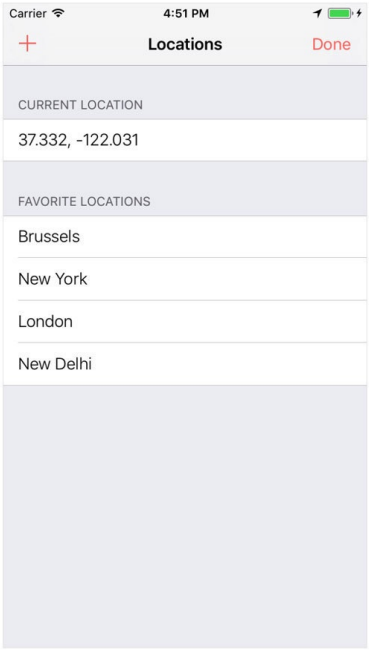


Data Flows in One Direction

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.

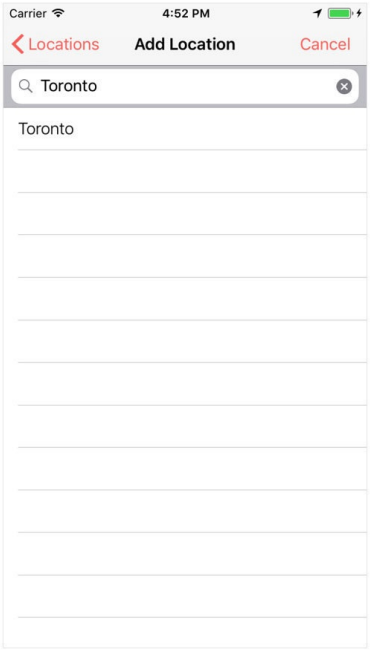
One of Cloudy's features is the ability for the user to search for and save locations. When the user selects one of the saved locations, Cloudy fetches weather data for that location and displays it to the user.

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.

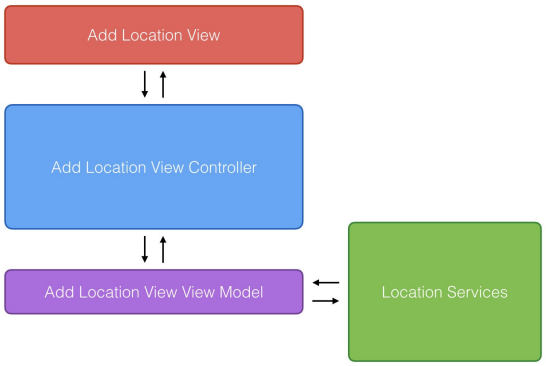
The user enters the name of the city she would like to add and Cloudy uses the **Core Location** framework to forward geocode the name of that

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

```
extension AddLocationViewController: UISearchBarDelegate {

    func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
        // Hide Keyboard
        searchBar.resignFirstResponder()

        // Forward Geocode Address String
        geocode(addressString: searchBar.text)
    }

    func searchBarCancelButtonClicked(_ searchBar: UISearchBar) {
        // Hide Keyboard
        searchBar.resignFirstResponder()

        // Clear Locations
        locations = []

        // Update Table View
        tableView.reloadData()
    }

}
```

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

```
extension AddLocationViewController: UITableViewDataSource {

    func tableView(_ tableView: UITableView, numberOfRowsInSection s\
ection: Int) -> Int {
        return locations.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath:\
 IndexPath) -> UITableViewCell {
        guard let cell = tableView.dequeueReusableCell(withIdentifie\
r: LocationTableViewCell.reuseIdentifier, for: indexPath) as? Locati\
onTableViewCell else { fatalError("Unexpected Table View Cell") }

        // Fetch Location
        let location = locations[indexPath.row]

        // Create View Model
        let viewModel = LocationsViewLocationViewModel(location: loc\
ation.location, locationAsString: location.name)

        // Configure Table View Cell
        cell.configure(withViewModel: viewModel)

        return cell
    }

}
```

The Core Location framework makes forward geocoding easy. The magic happens in the `geocode(addressString:)` method.

AddLocationViewController.swift

```
private func geocode(addressString: String?) {
    guard let addressString = addressString else {
        // Clear Locations
        locations = []

        // Update Table View
        tableView.reloadData()

        return
    }

    // Geocode City
    geocoder.geocodeAddressString(addressString) { [weak self] (plac\
emarks, error) in
        DispatchQueue.main.async {
            // Process Forward Geocoding Response
            self?.processResponse(withPlacemarks: placemarks, error:\
 error)
        }
    }
}
```

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

```
private func processResponse(withPlacemarks placemarks: [CLPlacemark\
]?, error: Error?) {
    if let error = error {
        print("Unable to Forward Geocode Address (\(error))")

    } else if let matches = placemarks {
        // Update Locations
        locations = matches.flatMap({ (match) -> Location? in
            guard let name = match.name else { return nil }
            guard let location = match.location else { return nil }
```

```
            return Location(name: name, latitude: location.coordinat\
e.latitude, longitude: location.coordinate.longitude)
        })

        // Update Table View
        tableView.reloadData()
    }
}
```

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

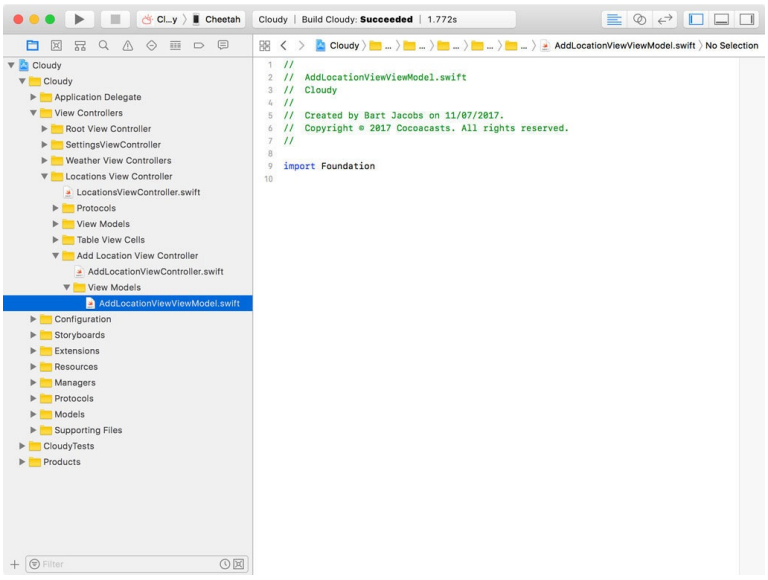
pattern, not understanding RxSwift. This also means that I won’t explain the details that relate to RxSwift in great detail. You can use any bindings solution you like, including a custom one.

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

```
import Foundation

class AddLocationViewViewModel {

}
```

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

```
import Foundation
import CoreLocation

class AddLocationViewViewModel {

    // MARK: - Properties

    private lazy var geocoder = CLGeocoder()

}
```

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

```
import Foundation
import CoreLocation

class AddLocationViewViewModel {

    // MARK: - Properties

    var query: String = ""

    // MARK: -

    private lazy var geocoder = CLGeocoder()

}
```

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

```
import Foundation
import CoreLocation

class AddLocationViewViewModel {

    // MARK: - Properties

    var query: String = ""


    // MARK: -

    private var querying: Bool = false
    private var locations: [Location] = []

    // MARK: -

    private lazy var geocoder = CLGeocoder()

}
```

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

```
import Foundation
import CoreLocation

class AddLocationViewViewModel {

    // MARK: - Properties

    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-value-observing 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

```
var query: String = "" {
    didSet {
        geocode(addressString: query)
    }
}
```

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

```
// MARK: - Helper Methods

private func geocode(addressString: String?) {
    guard let addressString = addressString, !addressString.isEmpty \
else {
        locations = []
        return
    }
}
```

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

```
// MARK: - Helper Methods

private func geocode(addressString: String?) {
    guard let addressString = addressString, !addressString.isEmpty \
else {
        locations = []
        return
```

```
    }

    querying = true

    // Geocode Address String
    geocoder.geocodeAddressString(addressString) { [weak self] (plac\
emarks, error) in

    }
}
```

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

```
// MARK: - Helper Methods

private func geocode(addressString: String?) {
    guard let addressString = addressString, !addressString.isEmpty \
else {
        locations = []
        return
    }

    querying = true

    // Geocode Address String
    geocoder.geocodeAddressString(addressString) { [weak self] (plac\
emarks, error) in
        var locations: [Location] = []

        self?.querying = false

        if let error = error {
            print("Unable to Forward Geocode Address (\(error))")

        } else {

        }
    }
}
```

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

```
// MARK: - Helper Methods

private func geocode(addressString: String?) {
    guard let addressString = addressString, !addressString.isEmpty \
else {
        locations = []
        return
    }

    querying = true

    // Geocode Address String
    geocoder.geocodeAddressString(addressString) { [weak self] (plac\
emarks, error) in
        var locations: [Location] = []

        self?.querying = false

        if let error = error {
            print("Unable to Forward Geocode Address (\(error))")

        } else if let _placemarks = placemarks {
            locations = _placemarks.flatMap({ (placemark) -> Locatio\
n? in
                guard let name = placemark.name else { return nil }
                guard let location = placemark.location else { retur\
n nil }
                return Location(name: name, latitude: location.coord\
inate.latitude, longitude: location.coordinate.longitude)
            })
        }

        self?.locations = locations
    }
}
```

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

```
// MARK: -

```

```
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

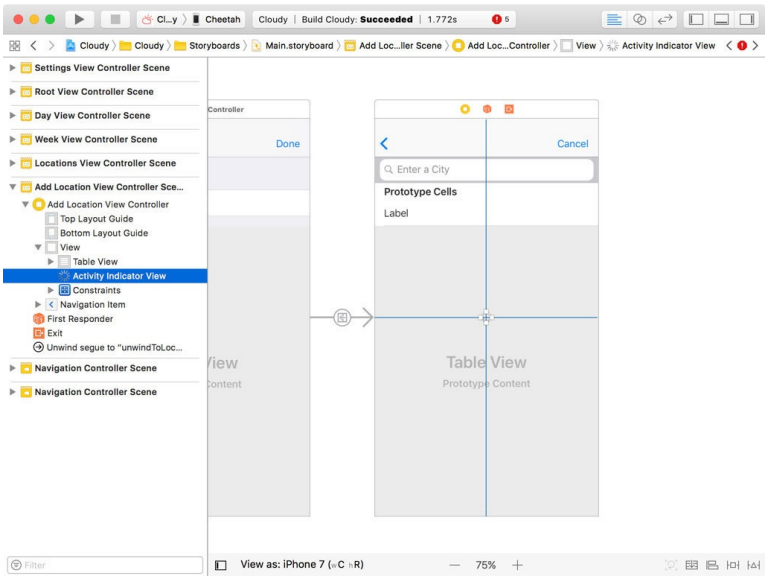
```
override func viewDidLoad() {
    super.viewDidLoad()

    // Set Title
    title = "Add Location"

     // Initialize View Model
     viewModel = AddLocationViewViewModel()

    // Configure View Model
    viewModel.locationsDidChange = { [unowned self] (locations) in
        self.tableView.reloadData()
    }

    viewModel.queryingDidChange = { [unowned self] (querying) in
        if querying {
            self.activityIndicatorView.startAnimating()
        } else {
            self.activityIndicatorView.stopAnimating()
        }
    }
}
```

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.

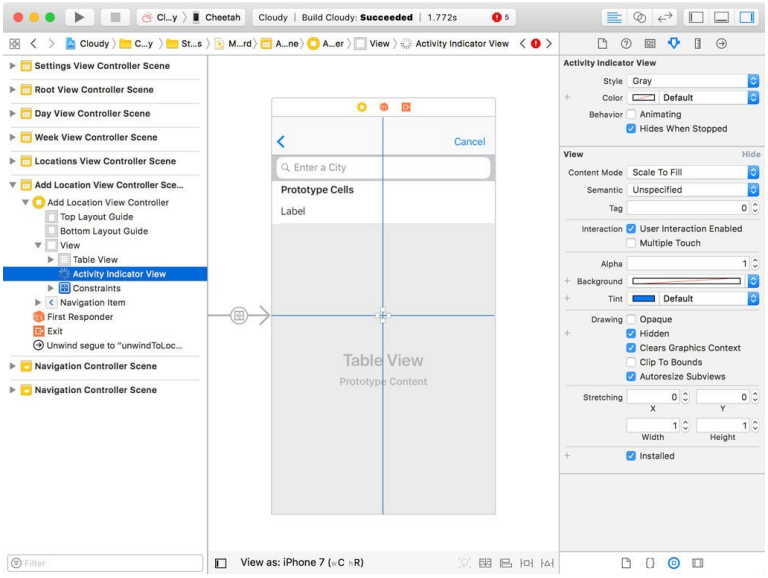
Updating the User Interface

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

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

```
extension AddLocationViewController: UITableViewDataSource {

    func tableView(_ tableView: UITableView, numberOfRowsInSection s\
ection: Int) -> Int {
        return viewModel.numberOfLocations
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath:\
 IndexPath) -> UITableViewCell {
        guard let cell = tableView.dequeueReusableCell(withIdentifie\
r: LocationTableViewCell.reuseIdentifier, for: indexPath) as? Locati\
onTableViewCell else { fatalError("Unexpected Table View Cell") }

        if let viewModel = viewModel.viewModelForLocation(at: indexP\
ath.row) {
            // Configure Table View Cell
            cell.configure(withViewModel: viewModel)
        }

        return cell
    }

}
```

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

```
extension AddLocationViewController: UITableViewDelegate {

    func tableView(_ tableView: UITableView, didSelectRowAt indexPat\
h: IndexPath) {
        guard let location = viewModel.location(at: indexPath.row) e\
lse { return }

        // Notify Delegate
        delegate?.controller(self, didAddLocation: location)

        // Pop View Controller From Navigation Stack
        navigationController?.popViewController(animated: true)
    }

}
```

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

```
extension AddLocationViewController: UISearchBarDelegate {

    func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
        // Hide Keyboard
        searchBar.resignFirstResponder()

        // Forward Geocode Address String
        viewModel.query = searchBar.text ?? ""
    }

    func searchBarCancelButtonClicked(_ searchBar: UISearchBar) {
        // Hide Keyboard
        searchBar.resignFirstResponder()

        // Forward Geocode Address String
        viewModel.query = searchBar.text ?? ""
    }

}
```

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 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.

```
target 'Cloudy' do
  platform :ios, '10.0'
  use_frameworks!

  pod 'RxSwift'
  pod 'RxCocoa'

  target 'CloudyTests' do
    inherit! :search_paths

    pod 'RxTest'
    pod 'RxBlocking'
  end
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.

```
Analyzing dependencies
Downloading dependencies
Installing RxBlocking (3.5.0)
Installing RxCocoa (3.5.0)
Installing RxSwift (3.5.0)
Installing RxTest (3.5.0)
Generating Pods project
Integrating client project

[!] Please close any current Xcode sessions and use `Cloudy.xcworksp\
ace` for this project from now on.
Sending stats
Pod installation complete! There are 4 dependencies from the Podfile\
 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

```
import RxSwift
import RxCocoa
import Foundation
import CoreLocation

class AddLocationViewViewModel {

    ...

}
```

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

```
// MARK: - Initializtion

init(query: Driver<String>) {

}
```

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

```
// MARK: - Initializtion

init(query: Driver<String>) {
    query
        .throttle(0.5)
        .distinctUntilChanged()
}
```

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

```
// MARK: - Initializtion

init(query: Driver<String>) {
    query
        .throttle(0.5)
        .distinctUntilChanged()
        .drive(onNext: { [weak self] (addressString) in
            self?.geocode(addressString: addressString)
        })
        .disposed(by: disposeBag)
}
```

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

```
var numberOfLocations: Int { return _locations.value.count }
```

AddLocationViewViewModel.swift

```
func location(at index: Int) -> Location? {
    guard index < _locations.value.count else { return nil }
    return _locations.value[index]
}
```

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

```
private func geocode(addressString: String?) {
    guard let addressString = addressString, !addressString.isEmpty \
else {
        _locations.value = []
        return
    }

    _querying.value = true

    // Geocode Address String
    geocoder.geocodeAddressString(addressString) { [weak self] (plac\
emarks, error) in
        var locations: [Location] = []

        self?._querying.value = false

        if let error = error {
            print("Unable to Forward Geocode Address (\(error))")

        } else if let _placemarks = placemarks {
            locations = _placemarks.flatMap({ (placemark) -> Locatio\
n? in
                guard let name = placemark.name else { return nil }
                guard let location = placemark.location else { retur\
n nil }
                return Location(name: name, latitude: location.coord\
inate.latitude, longitude: location.coordinate.longitude)
            })
        }

        self?._locations.value = locations
    }
}
```

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

```
import UIKit
import RxSwift
import RxCocoa

protocol AddLocationViewControllerDelegate {
    func controller(_ controller: AddLocationViewController, didAddL\
ocation location: Location)
}

class AddLocationViewController: UIViewController {

    ...

}
```

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

```
// MARK: -

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

```
// MARK: - View Life Cycle

```

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

    // Set Title
    title = "Add Location"

    // Initialize View Model
    viewModel = AddLocationViewViewModel(query: searchBar.rx.text.or\
Empty.asDriver())

    // Configure View Model
    viewModel.locationsDidChange = { [unowned self] (locations) in
        self.tableView.reloadData()
    }

    viewModel.queryingDidChange = { [unowned self] (querying) in
        if querying {
            self.activityIndicatorView.startAnimating()
        } else {
            self.activityIndicatorView.stopAnimating()
        }
    }
}
```

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

```
// Initialize View Model
viewModel = AddLocationViewViewModel(query: searchBar.rx.text.orEmpt\
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

```
// MARK: - View Life Cycle

override func viewDidLoad() {
    super.viewDidLoad()

    // Set Title
    title = "Add Location"

    // Initialize View Model
    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.

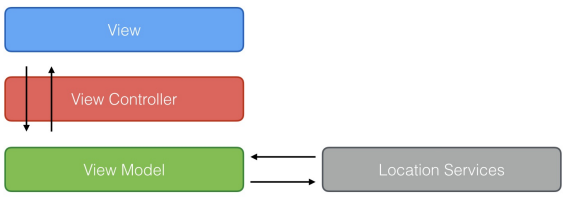
We could do the same for the `UITableViewDataSource` and `UITableViewDelegate` protocols, but I don't want to overwhelm you too much at this point. Build and run Cloudy to make sure we didn't break anything during the refactoring operation.

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.

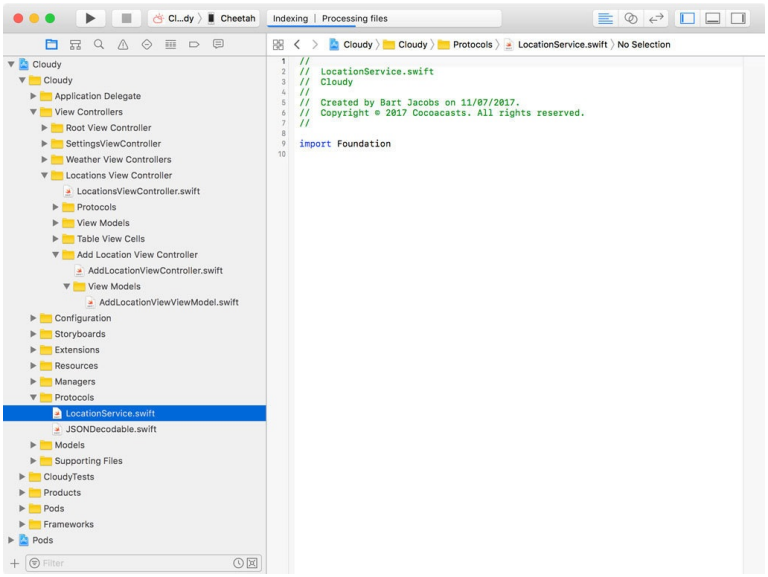
Not only does this solution decouple the view model from the Core Location framework, the view model won't even know which service it's using, that is, as long as the service conforms to the protocol we define.

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

```
protocol LocationService {

}
```

We first define a type alias, `LocationServiceCompletionHandler`. This is primarily for convenience.

LocationService.swift

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

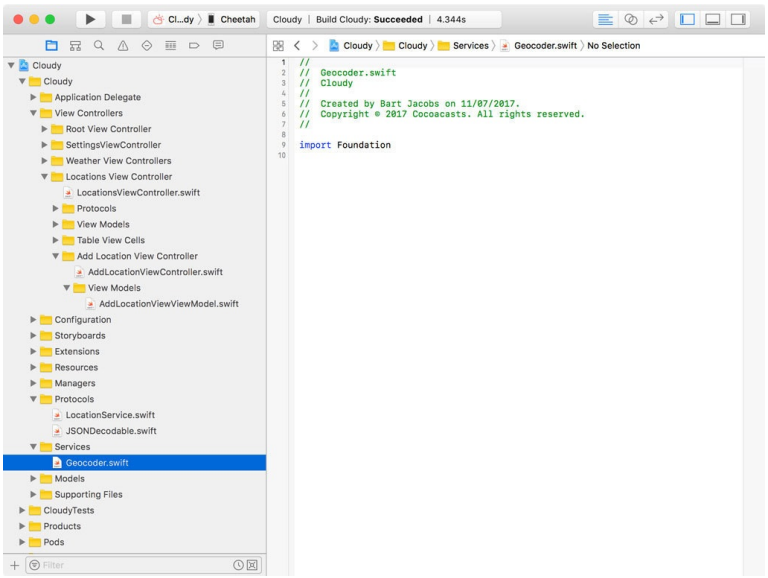
More important is the definition of the method that performs the geocoding request, `geocode(addressString:completionHandler:)`. It accepts an address string and a completion handler. Because the geocoding request is performed asynchronously, we mark the completion handler as escaping. The completion handler is of type `LocationServiceCompletionHandler`.

LocationService.swift

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

```
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 }
```

```
                guard let location = placemark.location else { retur\
n nil }
                return Location(name: name, latitude: location.coord\
inate.latitude, longitude: location.coordinate.longitude)
            })

            completionHandler(locations, nil)
        }
    }
}
```

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

```
// MARK: -

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

```
import RxSwift
import RxCocoa
import Foundation

class AddLocationViewViewModel {

    ...

}
```

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

```
// MARK: - Initialization

init(query: Driver<String>, locationService: LocationService) {
    // Set Properties
    self.locationService = locationService

    query
        .throttle(0.5)
        .distinctUntilChanged()
        .drive(onNext: { [weak self] (addressString) in
            self?.geocode(addressString: addressString)
        })
        .disposed(by: disposeBag)
}
```

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

```
private func geocode(addressString: String?) {
    guard let addressString = addressString, addressString.character\
s.count > 2 else {
        _locations.value = []
        return
    }

    _querying.value = true

    // Geocode Address String
    locationService.geocode(addressString: addressString) { [weak se\
lf] (locations, error) in
        self?._querying.value = false
        self?._locations.value = locations

        if let error = error {
            print("Unable to Forward Geocode Address (\(error))")
        }
    }
}
```

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

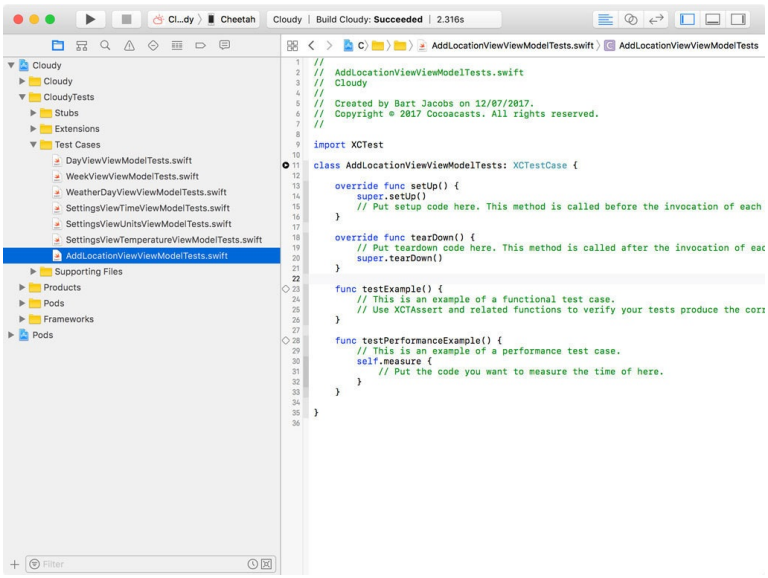
```
1 // Initialize View Model
2 viewModel = AddLocationViewViewModel(query: searchBar.rx.text.orEmpt\
3 y.asDriver(), locationService: Geocoder())
```

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

```
override func setUp() {
    super.setUp()

    // Initialize Query
    query = Variable<String>("")
}
```

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

```
import XCTest
import RxTest
import RxSwift
import RxBlocking
@testable import Cloudy

```

```
class AddLocationViewViewModelTests: XCTestCase {

    private class MockLocationService: LocationService {

    }

    ...

}
```

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

```
private class MockLocationService: LocationService {

    func geocode(addressString: String?, completionHandler: @escapin\
g LocationServiceCompletionHandler) {
        if let addressString = addressString, !addressString.isEmpty\
 {
            // Create Location
            let location = Location(name: "Brussels", latitude: 50.8\
503, longitude: 4.3517)

            // Invoke Completion Handler
            completionHandler([location], nil)
        } else {
            // Invoke Completion Handler
            completionHandler([], nil)
        }
    }

}
```

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

```
override func setUp() {
    super.setUp()

    // Initialize Query
    query = Variable<String>("")

    // Initialize Location Service
    let locationService = MockLocationService()

    // Initialize View Model
    viewModel = AddLocationViewViewModel(query: query.asDriver(), lo\
cationService: locationService)
}
```

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

```
override func setUp() {
    super.setUp()

    // Initialize Query
    query = Variable<String>("")

    // Initialize Location Service
    let locationService = MockLocationService()

    // Initialize View Model
    viewModel = AddLocationViewViewModel(query: query.asDriver(), lo\
cationService: locationService)

    // Initialize Scheduler
    scheduler = ConcurrentDispatchQueueScheduler(qos: .default)
}
```

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

```
// MARK: - Tests for Locations

func testLocations_HasLocations() {

}
```

We create an observable and store a reference to it in a constant named `observable`.

AddLocationViewViewModelTests.swift

```
// Create Subscription
let observable = viewModel.locations.asObservable().subscribeOn(sche\
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

```
// Set Query
query.value = "Brus"
```

We use `toBlocking()` to make the test synchronous. The result is stored in the `result` constant.

AddLocationViewViewModelTests.swift

```
// Fetch Result
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

```
XCTAssertNotNil(result)
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

```
// Fetch Location
let location = result.first!

XCTAssertEqual(location.name, "Brussels")
```

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

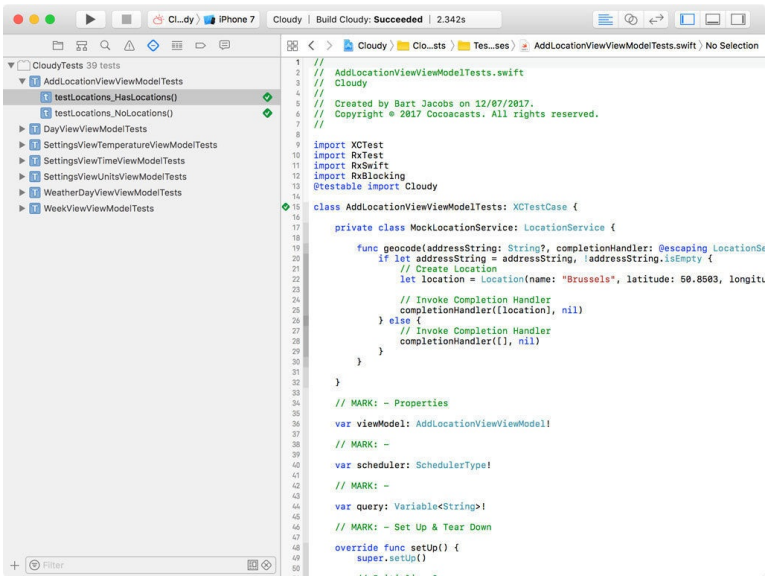
```
// MARK: - Tests for Locations

func testLocations_HasLocations() {
    // Create Subscription
    let observable = viewModel.locations.asObservable().subscribeOn(\
scheduler)

    // Set Query
    query.value = "Brus"

    // Fetch Result
    let result = try! observable.skip(1).toBlocking().first()!

    XCTAssertNotNil(result)
    XCTAssertEqual(result.count, 1)

    // Fetch Location
    let location = result.first!

    XCTAssertEqual(location.name, "Brussels")
}
```

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

```
func testLocations_NoLocations() {
    // Create Subscription
    let observable = viewModel.locations.asObservable().subscribeOn(\
scheduler)

    // Fetch Result
    let result: [Location] = try! observable.toBlocking().first()!

    XCTAssertNotNil(result)
    XCTAssertEqual(result.count, 0)
}
```

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

```
func testLocationAtIndex_Nil() {
    // Create Subscription
    let observable = viewModel.locations.asObservable().subscribeOn(\
scheduler)

    // Set Query
    query.value = "Brus"

    // Fetch Result
    let _ = try! observable.skip(1).toBlocking().first()!

    // Fetch Location
    let result = viewModel.location(at: 1)

    XCTAssertNil(result)
}
```

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.