# 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 2017-11-29


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.

# 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. If you want to follow along, make sure you have Xcode 9 installed on your machine.

## 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.

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. 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].

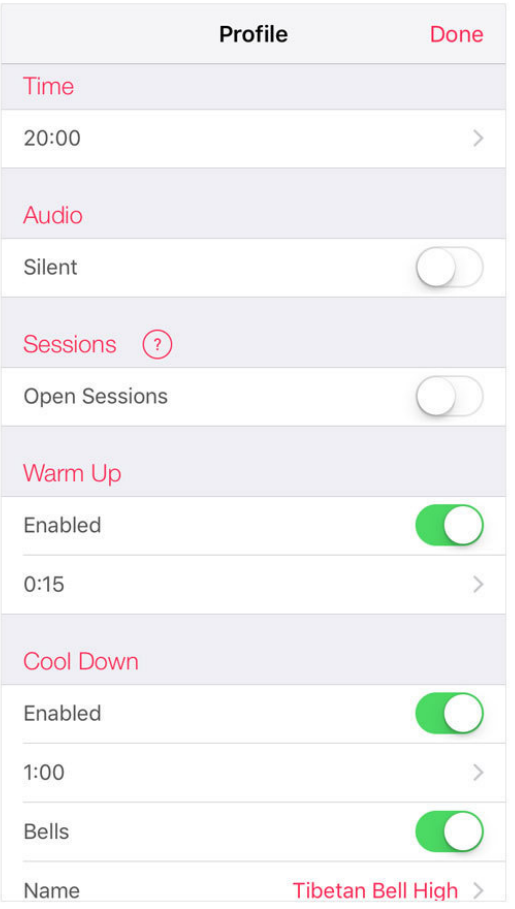
[1] https://cocoacasts.com/mastering-model-view-viewmodel-with-swift/

# 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[2], a meditation application I've been developing for the past few years. The settings view is an important aspect of the application.

[2] https://itunes.apple.com/app/samsara-meditation-yoga-timer/id592333521?mt=8




**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?

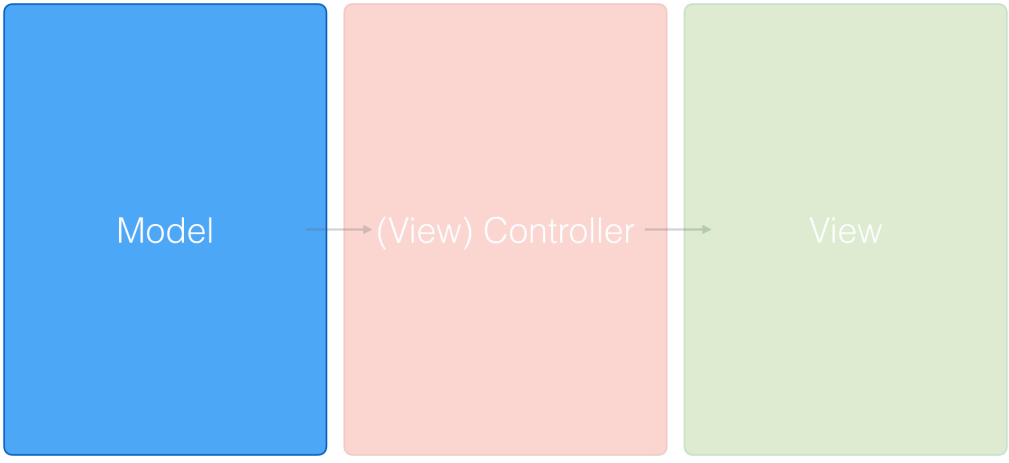
# What Is It?

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.



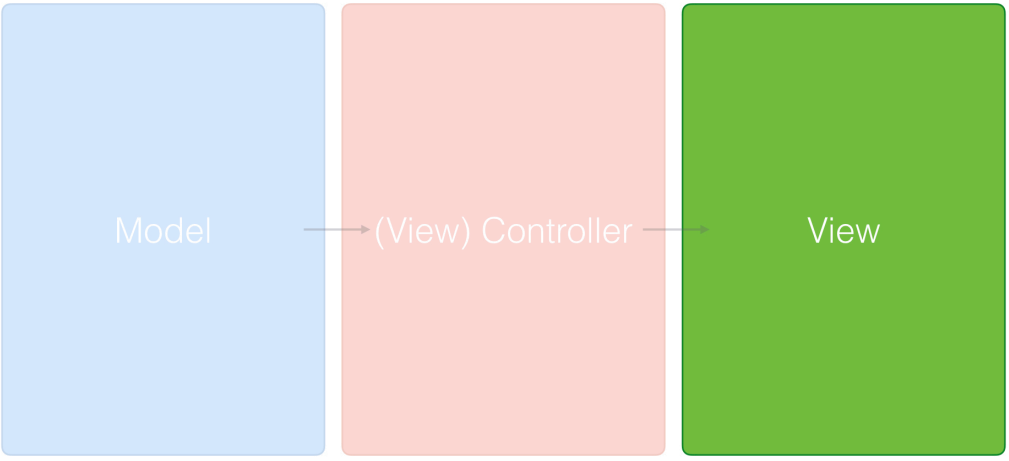

**The M In MVC**

## View

The view layer has two important tasks:

- presenting data to the user
- handling user interaction

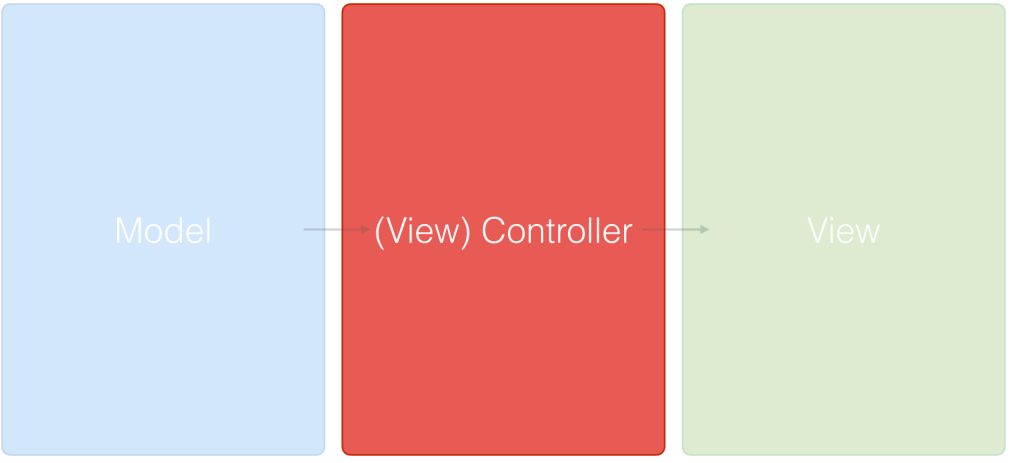
A core principle of the MVC pattern is the view layer's ignorance with respect to the model layer. Views are dumb objects. They only know how to present data to the user. They don't know or understand *what* they're presenting. This makes them flexible and easy to reuse.




**The V In MVC**

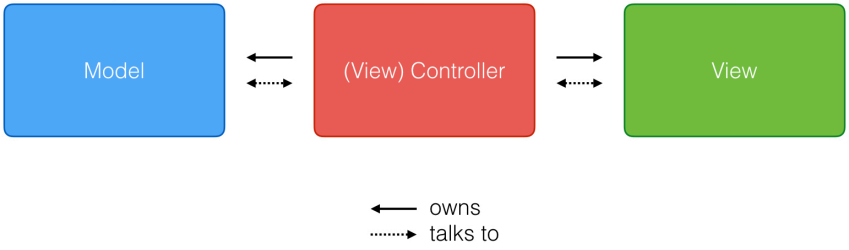
## Controller

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 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.

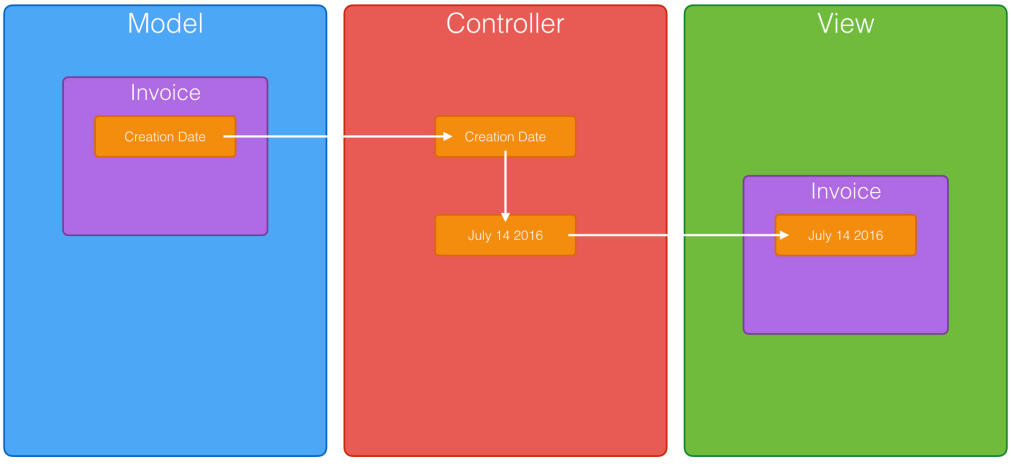
## Problems

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?

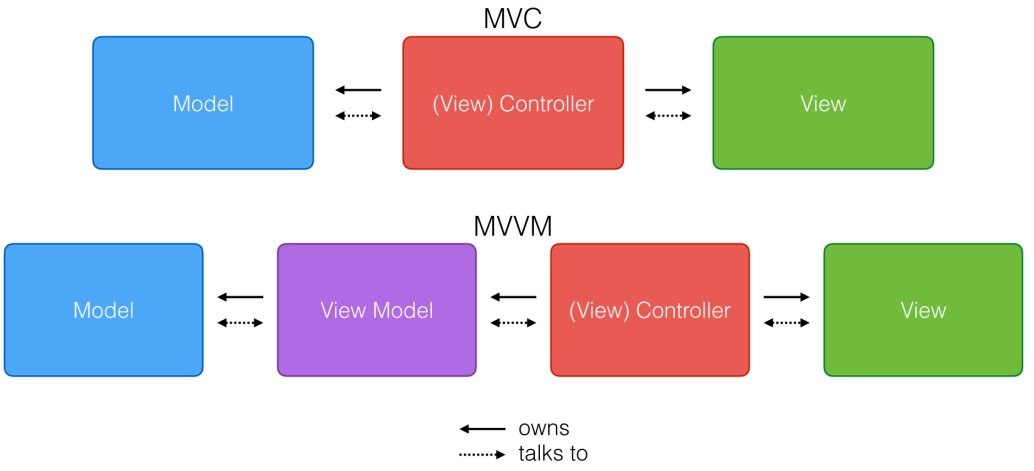
## How Can We Solve This?

In recent years, another pattern has been gaining traction in the Cocoa community. It's commonly referred to as the Model-View-ViewModel[3]

[3] https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel

pattern, MVVM for short. The origins of the MVVM pattern lead back to Microsoft's .NET[4] 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

[4] https://en.wikipedia.org/wiki/.NET_Framework

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.

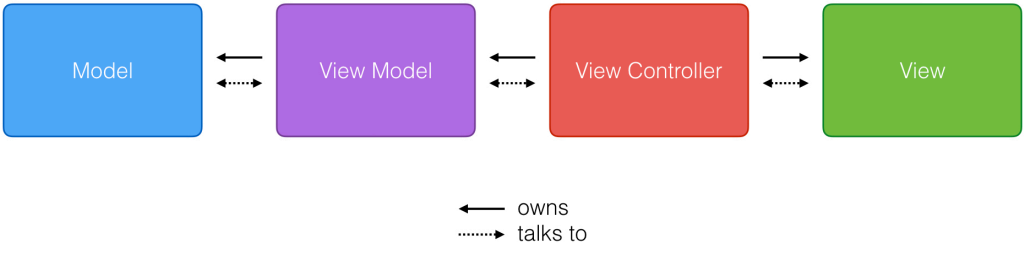
In the next chapter, we take a closer look at the internals of the Model-View-ViewModel pattern.

# 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

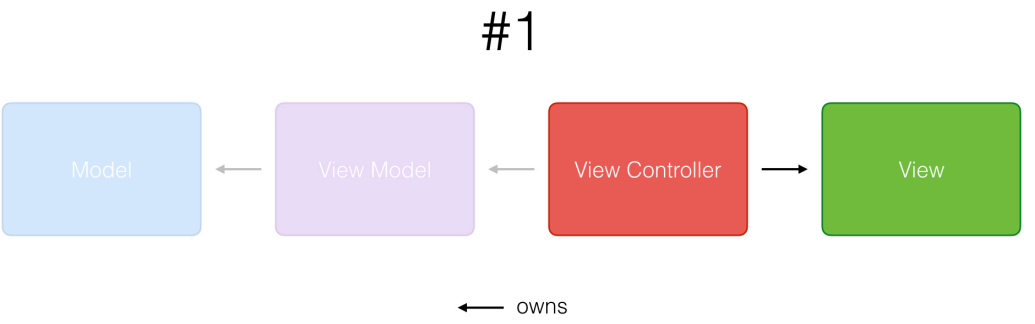
a transparent interface to the view controller, which it uses to populate the view layer and interact with the model layer. This results in a transparent communication between the four components or layers of your application.

## 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

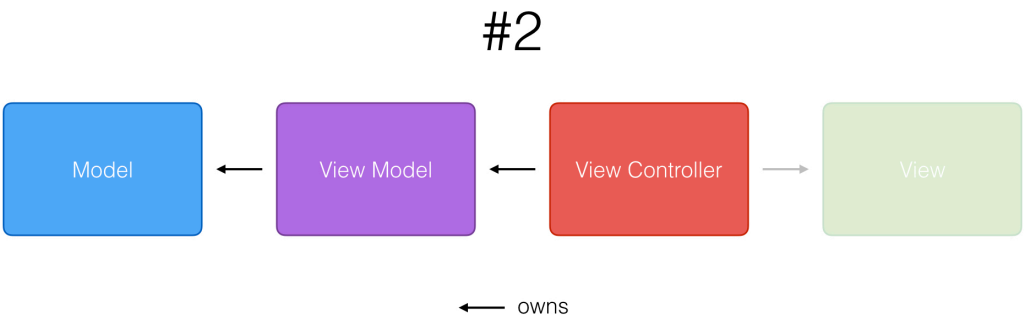
First, the view doesn't know about the view controller it's owned by. Remember that views are supposed to be dumb. They only know how to present what they're given by the view controller to the user. This is a rule you should never break. Ever.




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

## Rule #2

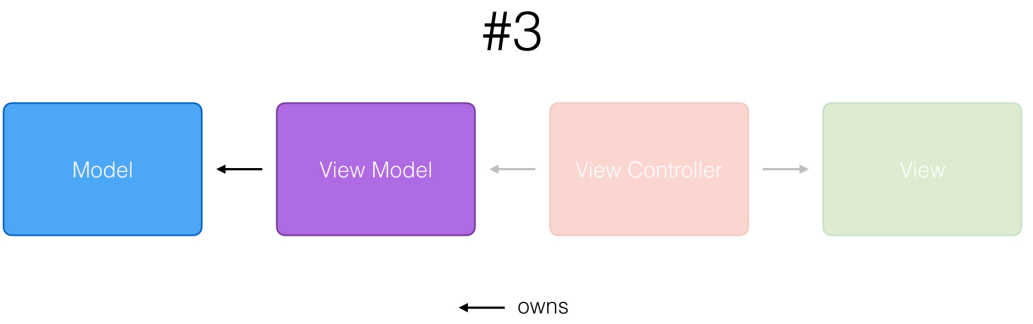
Second, the view or window controller doesn't know about the model. This is something that separates MVC from MVVM.




**The view controller doesn't know about the model.**

## Rule #3

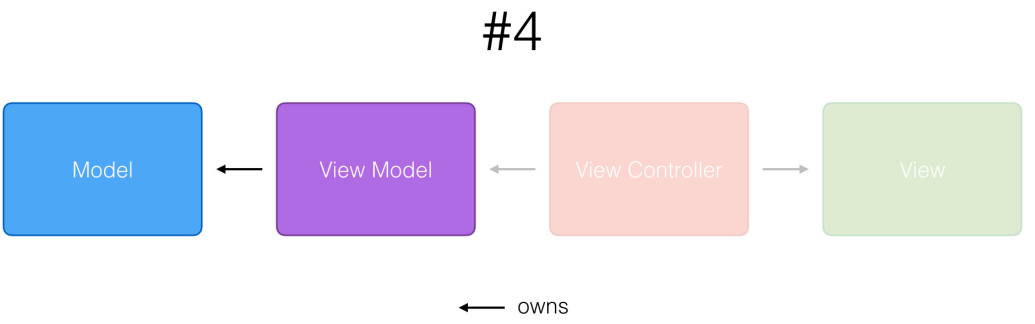
The model doesn't know about the view model it's owned by. This is another rule that should never be broken. The model should have no clue who it's owned by.




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

## Rule #4

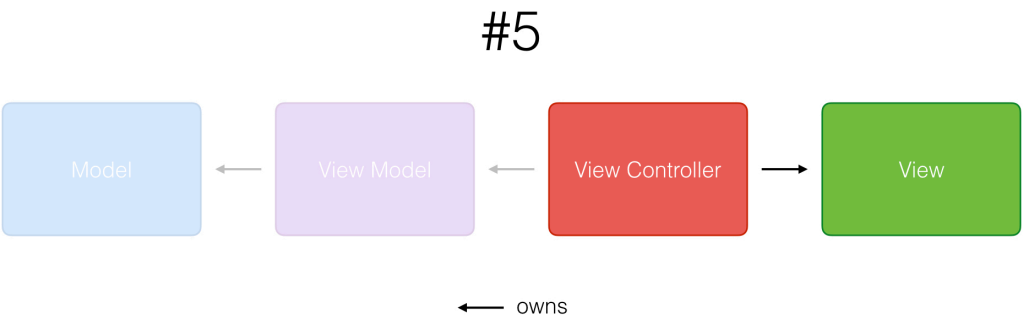
The view model owns the model. In a Model-View-Controller application, the model is usually owned by the view or window controller.




**The view model owns the model.**

## Rule #5

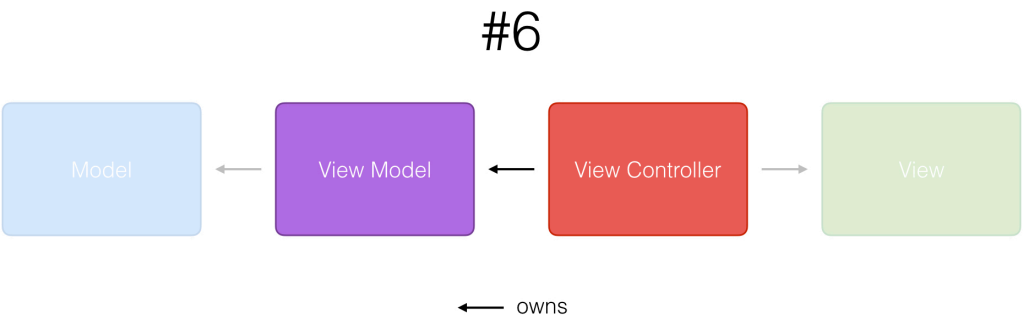
The view or window controller owns the view or window. This relationship remains unchanged.




**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.

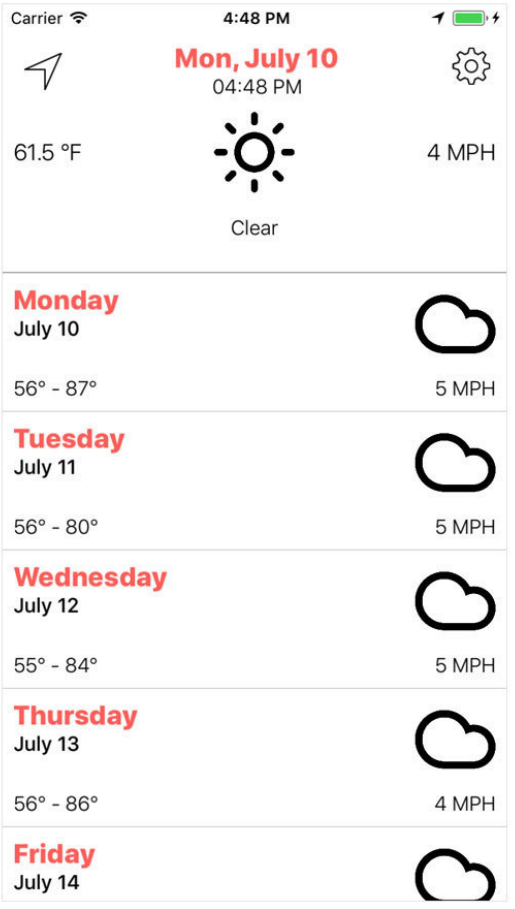
# 3 Meet Cloudy

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?

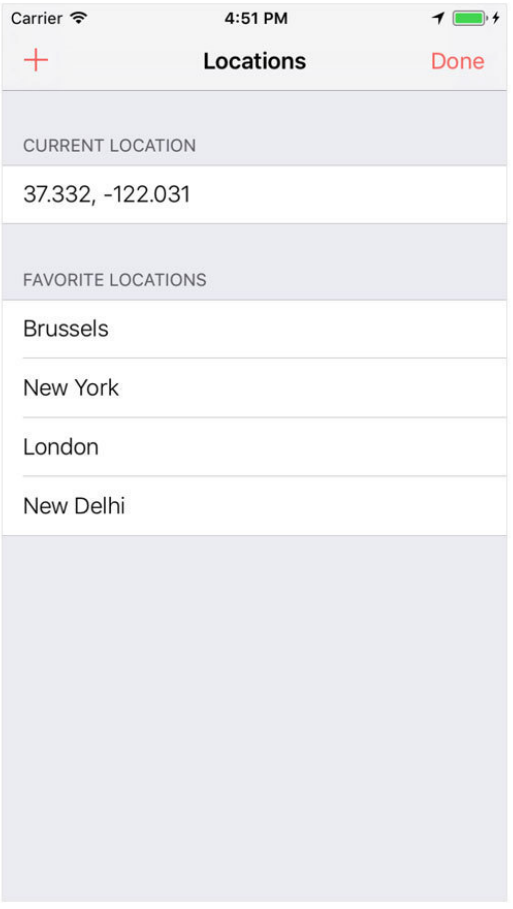
The application we're going to refactor is **Cloudy**. Cloudy is a lightweight weather application that shows the user the weather of their current location or a saved location. It shows the current weather conditions and a forecast for the next few days. The weather data is retrieved from the Dark Sky API[5], an easy-to-use weather service.

[5] https://darksky.net/dev/

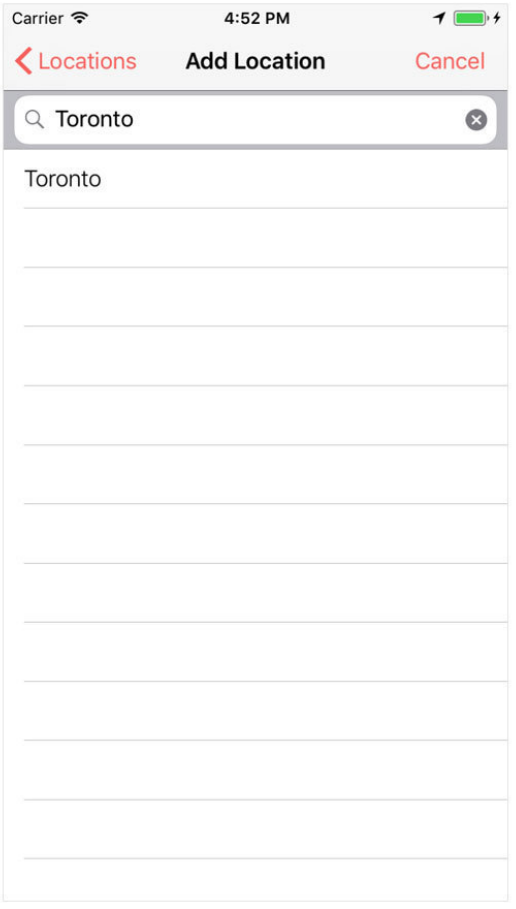



**Meet Cloudy**

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

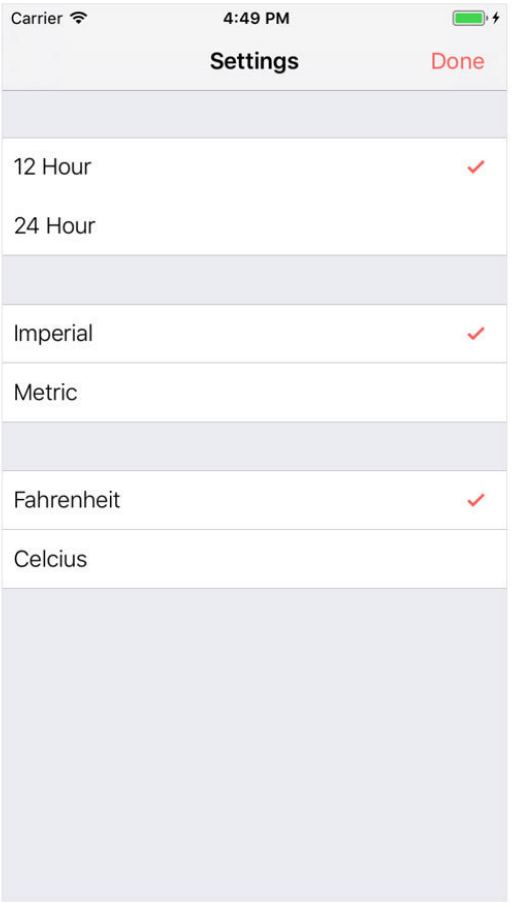



**Managing Locations**




**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**

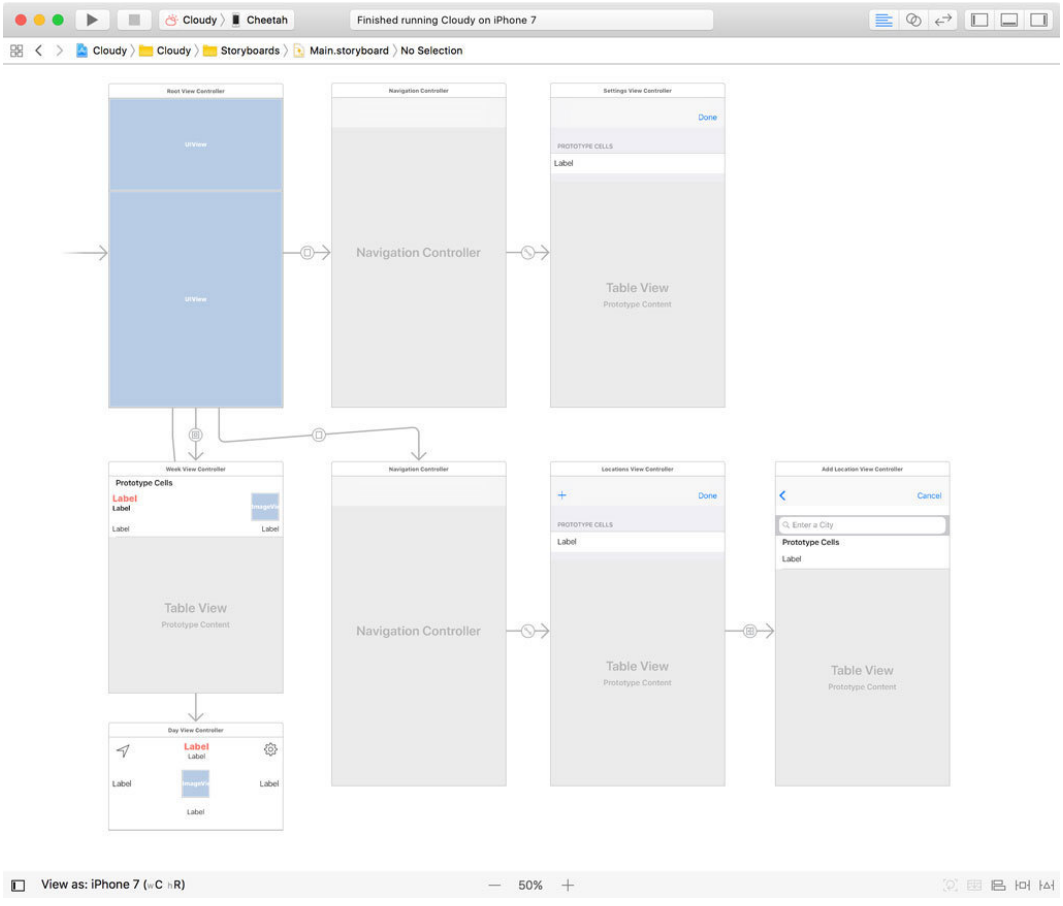
# Application Architecture

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

## Storyboard

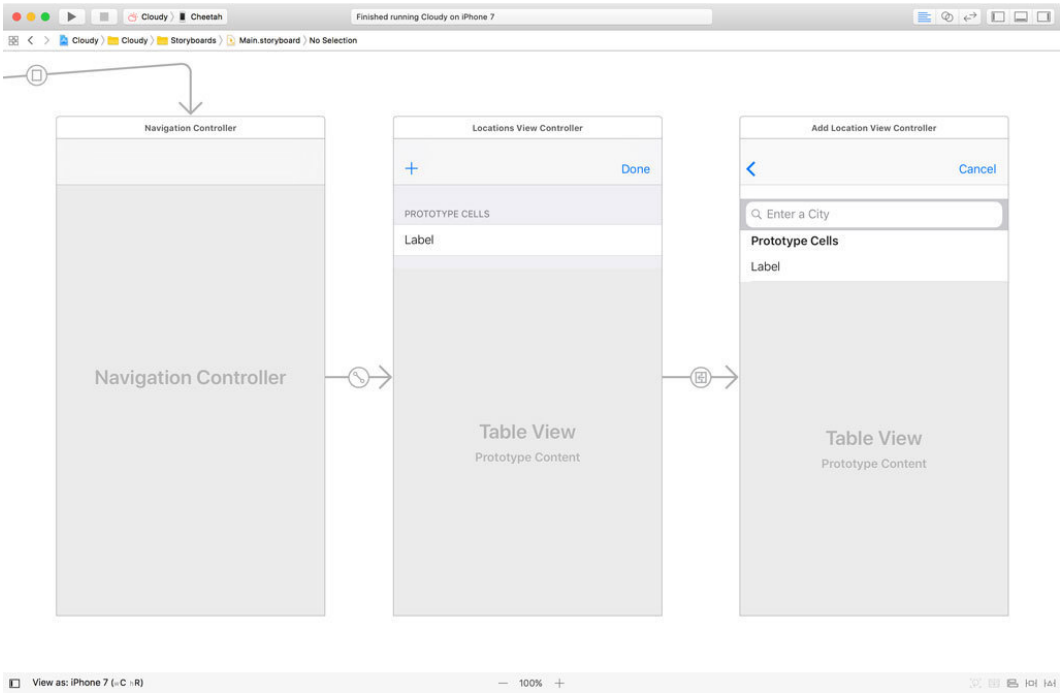
The main storyboard is the best place to start. You can see that we have a container view controller with two child view controllers. The top child

view controller shows the current weather conditions, the bottom child view controller displays the forecast for the next few days in a table view.



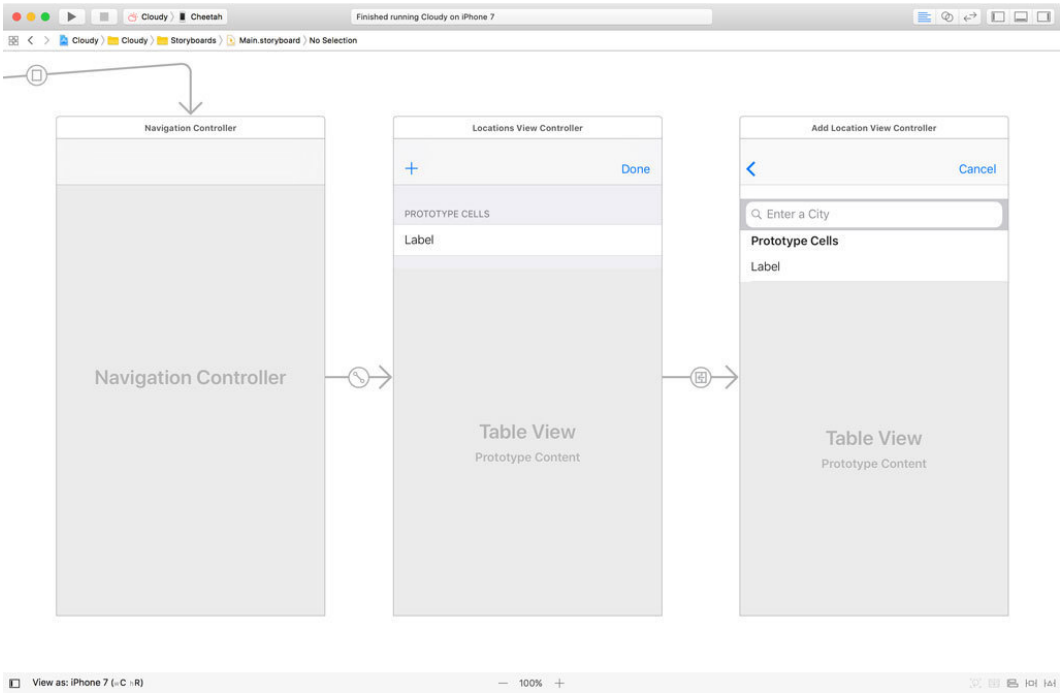

**Cloudy's Main Storyboard**

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**

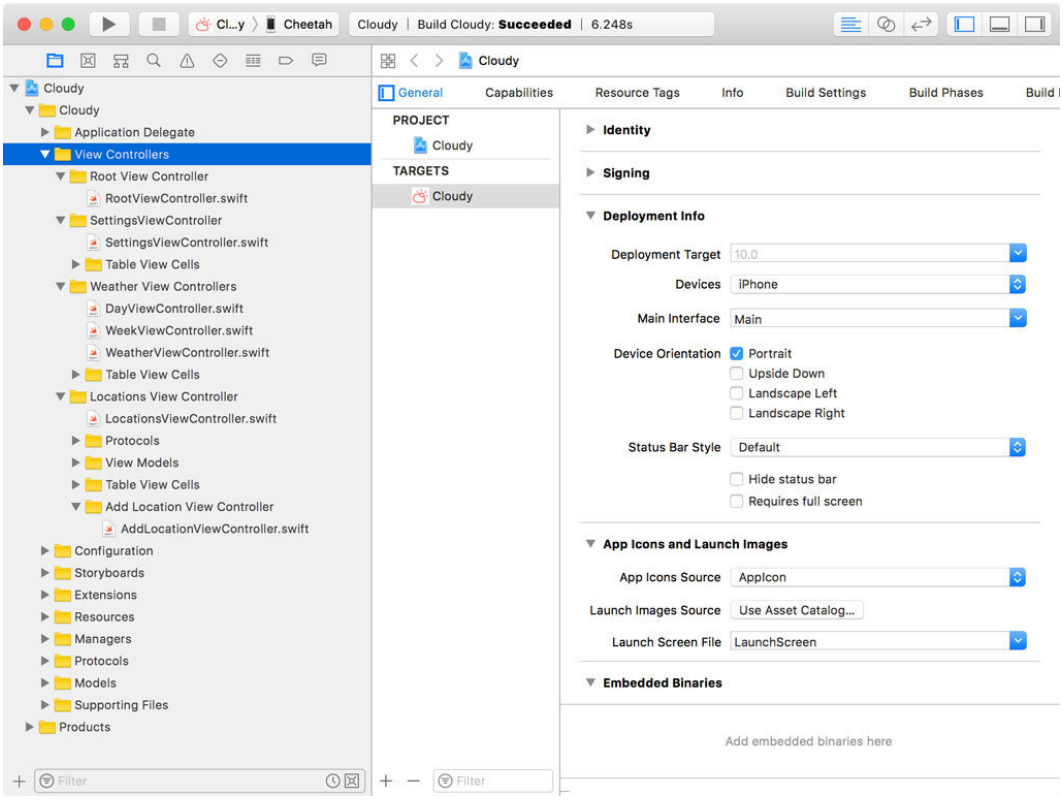
If the user taps the settings button in the top child view controller (top right), the settings view controller is shown. This is another table view listing the options we discussed earlier.




**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:complet` method of the `RootViewController` class, the weather data is sent to the day view controller and the week view controller.

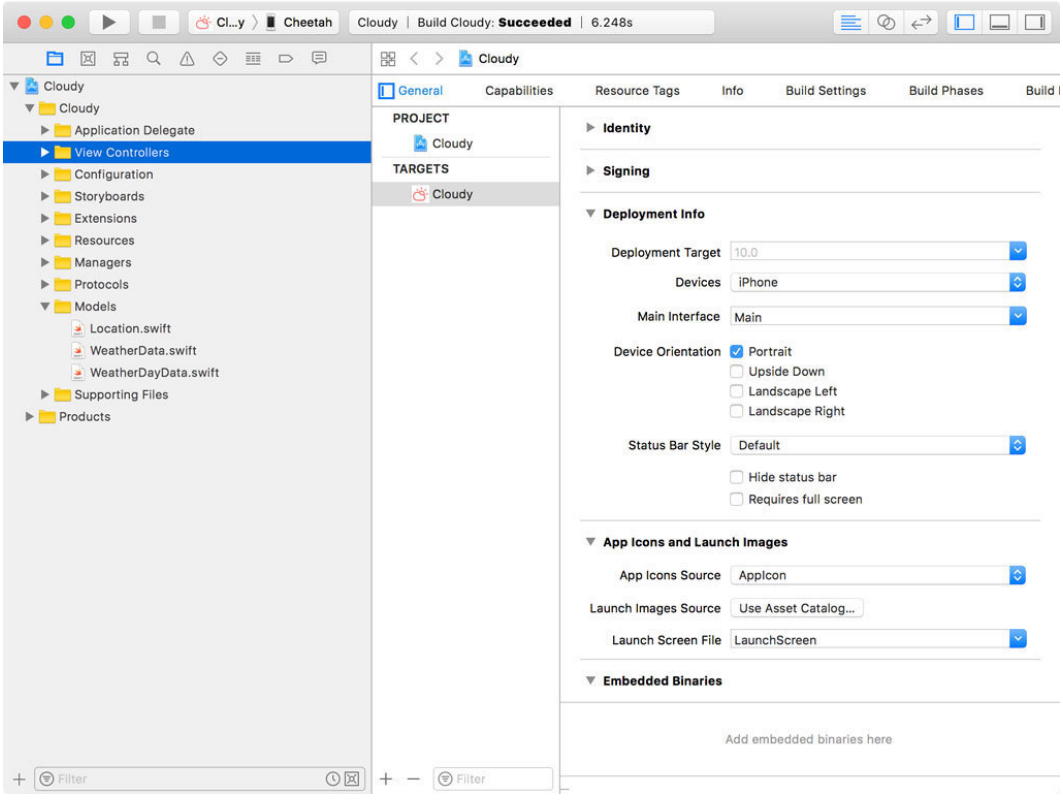
**RootViewController.swift**

```
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**

```
import Foundation

struct WeatherData {

    let time: Date

    let lat: Double
    let long: Double
    let windSpeed: Double
    let temperature: Double

    let icon: String
    let summary: String

    let dailyData: [WeatherDayData]

}
```

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)
    }
```

```
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  }
```

## 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 {
```

```
36              cell.windSpeedLabel.text = String(format: "%.f MPH", win\
37  dSpeed)
38          }
39
40          cell.iconImageView.image = imageForIcon(withName: weatherDat\
41  a.icon)
42      }
43
44      return cell
45  }
```

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()
    }

```

```
15      func controllerDidChangeTemperatureNotation(controller: Settings\
16  ViewController) {
17          dayViewController.reloadData()
18          weekViewController.reloadData()
19      }
20
21  }
```

## 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**

```
1   struct API {
2
3       static let APIKey = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
4       static let BaseURL = URL(string: "https://api.darksky.net/foreca\
5   st/")!
6
7       static var AuthenticatedBaseURL: URL {
8           return BaseURL.appendingPathComponent(APIKey)
9       }
10
11  }
```

# 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**

```
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)
```

```
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**

```
1  var week: [WeatherDayData]? {
2      didSet {
3          updateView()
4      }
5  }
```

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 {
```

```
36              cell.windSpeedLabel.text = String(format: "%.f MPH", win\
37  dSpeed)
38          }
39
40          cell.iconImageView.image = imageForIcon(withName: weatherDat\
41  a.icon)
42      }
43
44      return cell
45  }
```

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.