objc ↑↓

App Architecture

iOS Application Patterns in Swift

By Chris Eidhof, Matt Gallagher, and Florian Kugler

Version 1.0 (May 2018)

For more books, articles, and videos visit us at https://www.objc.io
Email: mail@objc.io
Twitter: @objcio

About This Book

This book is about application architecture: the structures and tools used to bring smaller components together to form an application. Architecture is an important topic in app development since apps typically integrate a large number of diverse components: user events, network services, file services, audio services, graphics and windowing services, and more. Integrating these components while ensuring state and state changes are reliably and correctly propagated between them requires a strong set of rules about how the components should interoperate.

Application Design Patterns

Sets of repeatedly applied design rules are called design patterns, and in this book, we will present an application that's fully implemented in five major application design patterns, which range from well established to experimental. These are:

→ Model-View-Controller (MVC)

→ Model-View-ViewModel+Coordinator (MVVM-C)

→ Model-View-Controller+ViewState (MVC+VS)

→ ModelAdapter-ViewBinder (MAVB)

→ The Elm Architecture (TEA)

The abstract block diagrams commonly used to describe application design patterns at the highest level do little to describe how these patterns are applied to iOS applications. To see what the patterns are like in practice, we'll take a detailed look at typical program flows in each of them.

We'll also look at a range of different patterns to show that there is no single best approach for writing programs. Any of the patterns could be the best choice depending upon the goals, desires, and constraints of you, your program, and your team. Application design patterns are not just a set of technical tools; they are also a set of aesthetic and social tools that communicate your application to you and to other readers of your code. As such, the best pattern is often the one that speaks most clearly to you.

Architectural Techniques and Lessons

After presenting each implementation, we'll spend some time discussing both the benefits each pattern offers for solving problems and how similar approaches can be used to solve problems in *any* pattern. As an example, reducers, reactive programming, interface decoupling, state enums, and multi-model abstractions are techniques often associated with specific patterns, but in this book, after we look at how these techniques are used within their patterns, we will also look at how their central ideas can solve problems across different patterns.

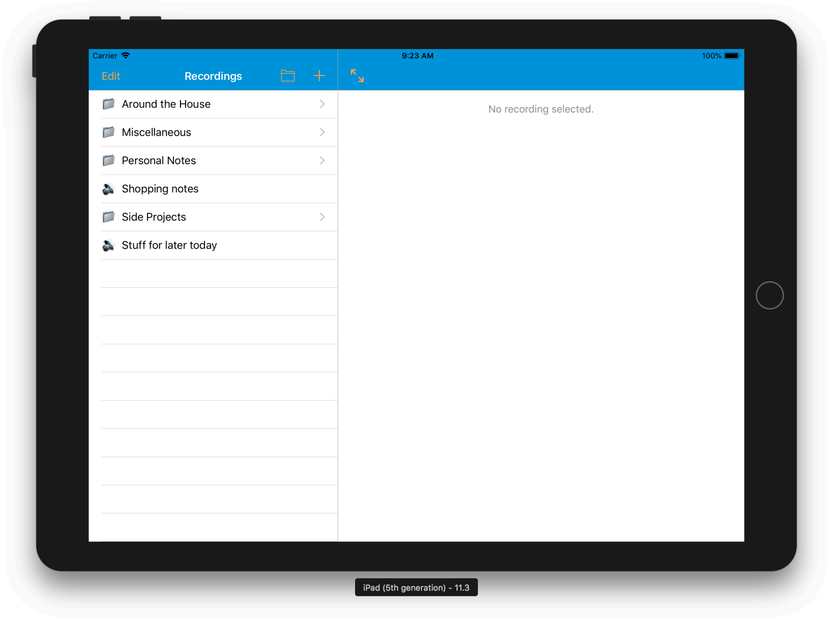
As this book will demonstrate, application architecture is a topic with multiple solutions to every problem. When properly implemented, all the solutions give the end user the same result. This means that ultimately, application architecture is about making choices to satisfy ourselves as programmers. We do this by asking questions such as what problems we want solved implicitly, what problems we want to consider on a case-by-case basis, where we need freedom, where we need consistency, where we want abstraction, where we want simplicity.

About the Recordings App

In this book, we show five different implementations of a single application: the Recordings app (the full source code of all implementations is available on GitHub). As the name might suggest, it's an app that records and plays voice notes. It also allows us to organize the recordings in a folder hierarchy. The app features a navigation stack of folder view controllers (for organizing files), a play view controller (for playing files), a modal record view controller presentation (for recording new files), and text alert controllers (for naming folders and recordings). All content is presented in a standard UIKit master-detail interface (the folder view controllers appear in the primary pane, and the play view controller appears in the secondary pane).

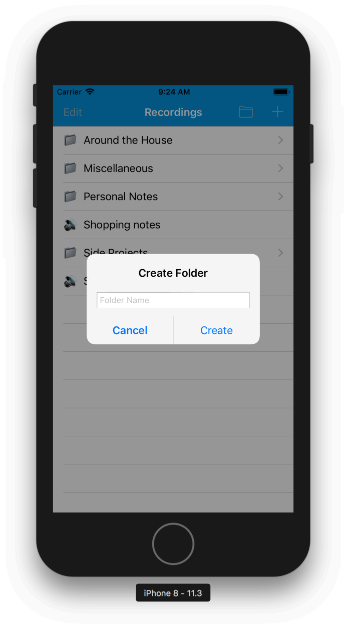
When choosing the sample app for this book, we made a list of criteria, and the Recordings app fulfills all of them. It is complex enough to show architectural patterns, yet simple enough to fit in a book. There is a nested navigation hierarchy. The app contains views with real-time updates, rather than just static views. The model is implemented as a persistent store that automatically saves each change to disk. We have two versions of the app that include networking. Our app works on both iPhone and iPad. Finally, we support state restoration.

Had we chosen a smaller app, it might have been easier to understand, but there would have been fewer opportunities to show differences between architectures. Had we chosen a larger app, the scalability of different architectural choices would have been more apparent, but the details would have been harder to see. We believe the Recordings app finds a good balance between these two extremes.

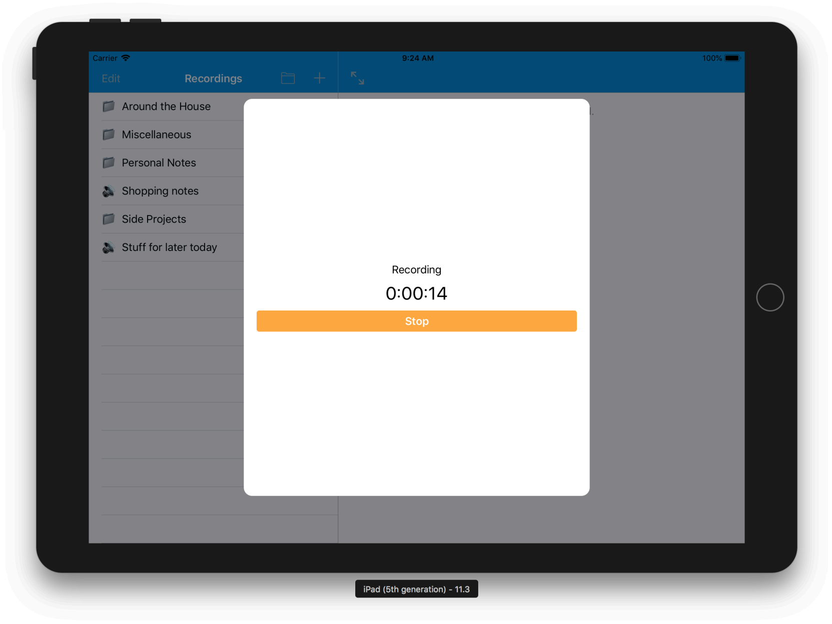


The app starts by showing a folder view controller for the root folder. The folder view controller is presented in a navigation controller in the primary pane of a split view controller. On the iPhone, this primary pane is presented in full-screen mode, and on an iPad, it is presented on the left side of the screen (this is standard behavior of a UISplitViewController on iOS).

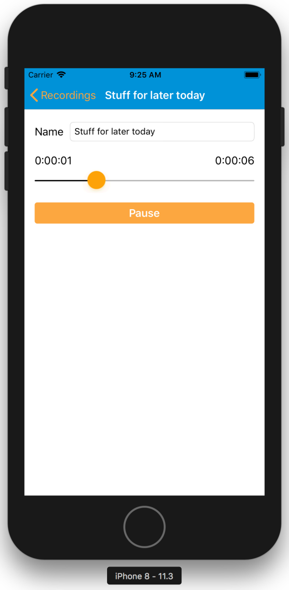
A folder can contain both recordings and nested folders. The folder view controller lets us add new folders and recordings and delete existing items.



Adding a folder brings up a modal alert view that asks for the folder name, and adding a recording immediately presents the record view controller modally.



When selecting an item in the folder view controller, the navigation action depends on the type of item: if we select a folder, a nested folder view controller is pushed onto the navigation stack, and if we select a recording, a play view controller is pushed onto the navigation stack.



The play view controller contains a text field used for both displaying and changing the name. It also contains a slider for the current play progress: as we're playing, the slider changes, and when we move the slider, the playback continues at the chosen position. Above the slider are labels for the current play position and total duration. Finally, the play view controller contains a button to start, pause, and resume playback.

Videos

We have recorded videos that go together with the book, and they are available as a separate purchase. The videos focus on topics that are best presented in a live coding and discussion format, such as:

→ Building a new feature in each of the five versions of the Recordings app (a mini player).

→ Building a very small app from scratch in eight different design patterns to highlight their commonalities and differences.

- Implementing a small version of the TEA framework.

We hope these videos will give you an even better idea of the practical implications of each of the application design patterns covered in this book.

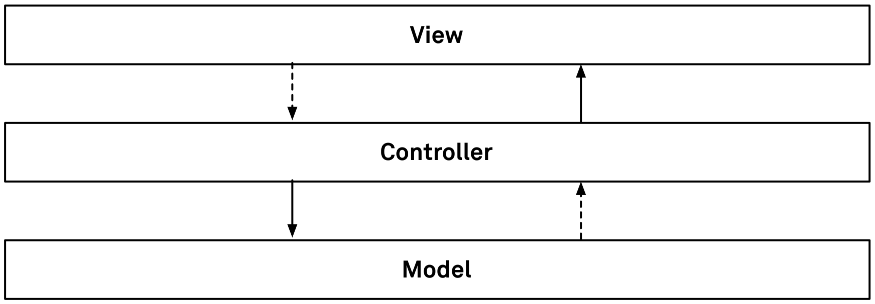
Introduction

1

Application Architecture

Application architecture is a branch of software design concerned with the structure of an application. More specifically, it looks at both how applications are divided into different interfaces and conceptual layers, and the control flow and data flow paths taken by different operations between and within these different components.

We often use simple block diagrams to explain an application's architecture. For example, Apple's Model-View-Controller (MVC) pattern describes three layers: the model, the view, and the controller layer.



The blocks in the above diagram show the different named layers of the pattern — the majority of the code written in an MVC project fits into one of these layers. The arrows show how the layers are connected.

However, a simple block diagram explains very little about how the pattern is expected to operate in practice. This is because an application architecture includes many expectations about how components should be constructed, how events flow through the layers, whether components should have compile-time or runtime references to each other, how data in different components should be read or mutated, and what paths state changes should take through the application structure.

Model and View

At the highest level, an application architecture is a basic set of categories into which different application components are separated. In this book, we call these different categories *layers*: a collection of interfaces and other code that conform to some basic rules and responsibilities.

The most common of these categories are the model layer and the view layer.

The *model layer* is an abstract description of the application's contents without any dependence on the application framework (such as UIKit). Therefore, it offers full control to the programmer. The model layer typically contains both model objects (examples in the Recordings app include folders and recordings) and coordinating objects (for example, objects that persist data on disk, such as the `Store` in our example application). The part of the model that's persisted to disk is called the *document model*.

The *view layer* is the application framework-dependent part that makes the model layer visible and user interactable, turning the model layer into an application. When writing iOS applications, the view layer almost always uses UIKit directly. However, as we will see, some architectures have a different view layer that wraps around UIKit. And for some custom applications, most notably games, the view layer might not be UIKit or AppKit; it could be SceneKit or a wrapper around OpenGL.

Sometimes, model or view instances are represented by structs or enums rather than objects. The difference is important in practice, but when we talk about objects, structs, and enums in the model layer, we'll call all three of them *model objects* — likewise for *view objects*, which might manifest as objects, structs, or enums as well.

View objects typically form a single *view hierarchy*, in which all objects are connected in a tree structure with the screen at the trunk, windows within the screen, and increasingly smaller views at the branches and leaves. Likewise, view controllers typically form a *view controller hierarchy*. Meanwhile, model objects don't necessarily form a hierarchy — there could be models in the program with no connection between them.

When we write *view*, we usually mean a single view object, such as a button or a label. When we write *model*, we usually mean a single model object, such as a `Recording` or `Folder` instance. In most literature on the subject, "the model" means different things depending on the context. It could mean the model layer, the concrete objects that are in the model layer, the document model, or a discrete document within the model layer. At the cost of verbosity, we try in this book to be explicit about the different meanings.

Why Are the Categories of Model and View Considered So Fundamental?

It is certainly possible to write an application where there's no separation between the model and view layers. As an example, in a simple modal dialog, there is often no separate model data. Instead, we read the state directly from user interface elements when the "OK" button is tapped. In general though, without a separated model layer, it's difficult to ensure that actions in a program occur according to any coherent rules.

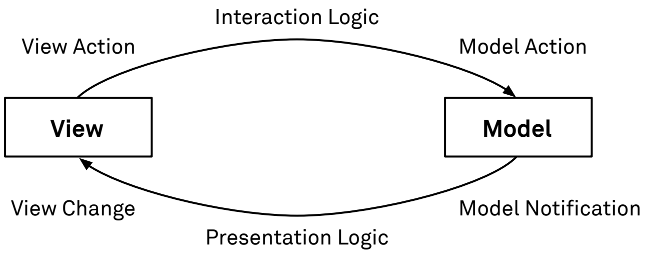
The most important reason to define a model layer is to have a single source of truth in our program that is clean and well behaved and not governed by the implementation details of the application framework.

An *application framework* provides infrastructure upon which we can build applications. In this book we use Cocoa — more specifically, UIKit, AppKit, or WatchKit, depending on the target platform — as the application framework.

If the model layer is kept separate from the application framework, we can use it completely outside the bounds of the application. We can easily run it in a separate testing harness, or we can write a new view layer using a different application framework and use our model layer in the Android, macOS, or Windows version of the application.

Applications Are a Feedback Loop

The view layer and the model layer need to communicate. There are, therefore, connections between the two. Assuming the view layer and model layer are clearly separated and not inextricably linked, communication between the two will require some form of translation:



The result is a feedback loop, which is not surprising, since a user interface with both display and input functionality is fundamentally a feedback device. The challenge for each application design pattern is how to handle the communication, dependencies, and transformations inherent in the arrows shown in this diagram.

Different parts of the path between the model layer and view layer have different names. The name *view action* refers to the code path triggered by a view in response to a user-initiated event, such as a tapping a button or selecting a row in a table view. When a view action is sent through to the model layer, it may be converted into a *model action* (an instruction to a model object to perform an action or update). This instruction might also be called a *message* (particularly when the model is changed using a *reducer*). The transformation of view actions into model actions and other logic along this path is called *interaction logic*.

A *model update* is a change to the state of one or more of the model objects. A model update usually triggers a *model notification*, i.e. an observable notification coming from the model layer that describes what has changed. When views are dependent on model data, notifications should trigger a *view change* to update the contents of the view layer. These notifications can take multiple forms: Foundation's `Notification`, delegates, callbacks, or another mechanism. The transformation of model notifications and data into view changes and other logic along this path is called *presentation logic*.

Depending on the application pattern, some state may be maintained outside the document model, and therefore actions that update that state do not follow a path through the document model. A common example of this in many patterns is the *navigation state*, where subsections of the view hierarchy (called *scenes*, following the terminology used by Cocoa storyboards) may be swapped in and out.

The state of an app that is not part of the document model is called *view state*. In Cocoa, most view objects manage their own view state, and controller objects manage the remaining view state. Diagrams of view state in Cocoa typically involve shortcuts across the feedback loop or individual layers that loop back on themselves. In other architectures, the view state is not part of the controller layer, but rather part of the model layer (however, by definition, view state isn't part of the *document model*).

When all state is maintained in the model layer and all changes follow this full feedback loop path, we call it *unidirectional data flow*. If the only way for any view object or intermediate layer object to be created or updated is via notifications from the model (i.e. there is no shortcut where a view or intermediate layer can update itself or another view), the pattern is usually unidirectional.

Architectural Technologies

The standard Cocoa frameworks on Apple's platforms provide some architectural tools. *Notifications* broadcast values from a single source to zero or more listeners. *Key-value observing* (KVO) can report changes to a property on one object to another object. However, because the list of architectural tools in Cocoa runs out quickly, we'll make use of additional frameworks.

One of the third-party technologies used in this book is *reactive programming*. Reactive programming is another tool for communicating changes, but unlike notifications or KVO, it focuses on the transformations between the source and destination, allowing logic to be expressed in transit between components.

We can use techniques like reactive programming or KVO to create bindings. A binding takes a source and a target, and whenever the source changes, it updates the target. This is syntactically different from a manual observation; rather than writing observation logic, we only specify source and target, and the framework takes care of the rest.

Cocoa on macOS includes Cocoa bindings, which is a two-way form of binding — all observables are also observers, and establishing a binding connection in one direction also establishes a connection in the reverse direction. None of the bindings provided by RxCocoa (used in the MVVM-C chapter) or CwlViews (used in the MAVB chapter) are two-way bindings — so any discussion of bindings in this book will refer solely to one-way bindings.

Application Tasks

For a program to function, views must be created, populated with model data, configured to make changes on the model, and updated when the model updates.

For this reason, we have to decide how to perform the following tasks:

1. **Construction** — Who constructs the model and the views and connects the two?

2. **Updating the model** — How are view actions handled?
3. **Changing the view** — How is model data applied to the view?
4. **View state** — How are navigation and other non-model state handled?
5. **Testing** — What testing strategies are used to achieve reasonable test case code coverage?

The answers to these five questions form the basis of the application design patterns we'll look at in this book.

Overview of Application Design Patterns

2

This book focuses on five different implementations of the Recordings application using the following application design patterns.

The first two are patterns in common use on iOS:

→ **Standard Cocoa Model-View-Controller** (MVC) is the pattern used by Apple in its sample applications. This is easily the most common architecture in Cocoa applications, and it's the baseline for any discussion about architecture in Cocoa.

→ **Model-View-ViewModel+Coordinator** (MVVM-C) is a variant of MVC with a separate view-model and a coordinator to manage the view controller hierarchy. MVVM uses data binding (typically with reactive programming) to establish the connections between the view-model and the view layer.

The other three patterns we look at are experimental architectures that are uncommon in Cocoa. We believe they offer useful insights into application architecture that can help improve our code, regardless of whether or not we adopt the entire architecture:

→ **Model-View-Controller+ViewState** (MVC+VS) centralizes the entire view state into a single location, following the same rules as the model, rather than spreading it among views and view controllers.

→ **ModelAdapter-ViewBinder** (MAVB) is an experimental architecture pattern by one of the authors. MAVB focuses on declarative view construction and uses bindings rather than controllers to communicate between the model and the view.

→ **The Elm Architecture** (TEA) is the most radical departure from common design patterns like MVC or MVVM. It uses a virtual view hierarchy to construct views and reducers that interact between models and views.

In this chapter, we'll give an overview of the philosophy and choices behind each of these patterns, while subsequent chapters will look at the impact of those choices on the implementation of the Recordings app.

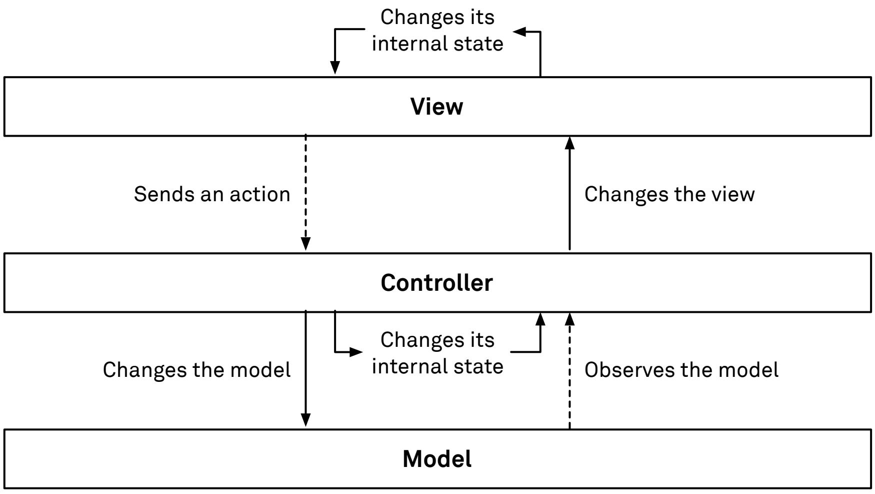
These five patterns are certainly not an exhaustive list of application design patterns for iOS, but as we discuss each one, we'll try to shed more light on why we think it's worth covering in the book. At the end of this chapter, we'll briefly discuss some of the patterns we omitted.

Model-View-Controller

In Cocoa MVC, a small number of controller objects handle all tasks that fall outside the model or view layers.

This means that the controller layer receives all view actions, handles all interaction logic, dispatches all model actions, receives all model notifications, prepares all data for presentation, and applies changes to the views. If we look at the diagram of the application feedback loop in the Introduction chapter, the controller is every labeled point on both arrows between model and view, in addition to construction and navigation tasks that aren't labeled.

The following is a block diagram of MVC showing the major communication paths through an MVC application:



The dotted lines in this diagram represent runtime references; neither the view layer nor the model layer reference the controller layer directly in code. Solid lines represent compile-time references; a controller instance knows the interface of the view and model objects to which it's connected.

If we trace the boundary on the outside of this diagram, we get the MVC version of the application feedback loop. Notice there are other paths through the diagram that don't take this whole path (indicated by the arrows that curve back on the view and controller layers).

1. Construction

The application object starts construction of top-level view controllers, which load and configure views and include knowledge about which data from the model must be represented. A controller either explicitly creates and owns the model layer, or it attempts to access the model through a lazily constructed singleton. In multi-document arrangements, the model layer is owned by a lower-level controller like `UIDocument`/`NSDocument`. Cached references to individual model objects relevant to views are usually held by the controllers.

2. Updating the Model

In MVC, the controller receives view events mostly through the target/action mechanism and delegates (set up in either storyboards or code). The controller knows what kind of views it's connected to, but the view has no static knowledge of the controller's interface. When a view event arrives, the controller can change its internal state, change the model, or directly change the view hierarchy.

3. Changing the View

In our interpretation of MVC, the controller should not directly change the view hierarchy when a model-changing view action occurs. Instead, the controller is subscribed to model notifications and changes the view hierarchy once a model notification arrives. This way, the data flows in a single direction: view actions get turned into model changes, and the model sends notifications that get turned into view changes.

4. View State

View state may be stored in properties on the view or the controller as needed. View actions affecting state on a view or controller are not required to pass via the model. View state can be persisted using a combination of support from the storyboard layer —

which records the active controller hierarchy — and through the implementation of `UIStateRestoring`, which is used to read data from the controllers and views.

5. Testing

In MVC, view controllers are tightly integrated with other components of an application. This lack of boundaries makes unit and interface tests difficult to write, leaving integration testing as one of the few viable testing approaches. In an integration test, we build connected sections of the view, model, and controller layers; we manipulate either the model or the view; and we test for the desired effect.

An integration test is complicated to write but also covers a lot of ground. It tests not just logic, but also how components are connected — although not to the same degree as UI tests do in some cases. However, it is usually possible to achieve around 80 percent test coverage using integration tests in MVC.

Importance of Model-View-Controller

As the pattern used by Apple in all sample applications — and the pattern Cocoa is designed to support — Cocoa MVC is the authoritative default application architectural pattern on iOS, macOS, tvOS, and watchOS.

The specific implementation of the Recordings app presented in this book offers an interpretation of MVC that we feel most accurately reflects a common denominator across the history of iOS and macOS. However, as we'll see later on, the freedom of the MVC pattern permits a large number of variants: many ideas from other patterns can be integrated within smaller sections of the overall app.

History

The name MVC was first used in 1979 by Trygve Reenskaug to describe the existing application "template pattern" in Smalltalk-76, following a terminology discussion with Adele Goldberg (previous names included Model-View-Editor and Model-View-Tool-Editor).

In the original formulation, views were directly "attached" to model objects (observing all changes), and the purpose of a controller was merely to capture user events and forward them to the model. Both of these traits are products of how Smalltalk worked and have little purpose in modern application frameworks, so the original formulation of MVC is rarely used today.

The enduring concepts from the original Smalltalk implementation of MVC are the premise of *separated presentation* — that the view and model layers should be kept apart — and that there's a strong need for a supporting object to aid the communication between the two.

The Cocoa implementation of MVC has its roots in NeXTSTEP 4 from approximately 1997. Prior to that time, all of the roles now handled by the controller were typically the responsibility of a high-level view class — often the `NSWindow`. The controller concept in NeXTSTEP is very similar to the presenter class from Taligent's earlier Model-View-Presenter. In modern contexts, the name Model-View-Presenter is often used for MVC-like patterns where the view is abstracted from the controller by a protocol.

Model-View-ViewModel+Coordinator

MVVM, like MVC, is structured around the idea of a scene — a subtree of the view hierarchy that may be swapped in or out during a navigation change.

The defining aspect of MVVM, relative to MVC, is that it uses a *view-model* for each scene to describe the presentation and interaction logic of the scene.

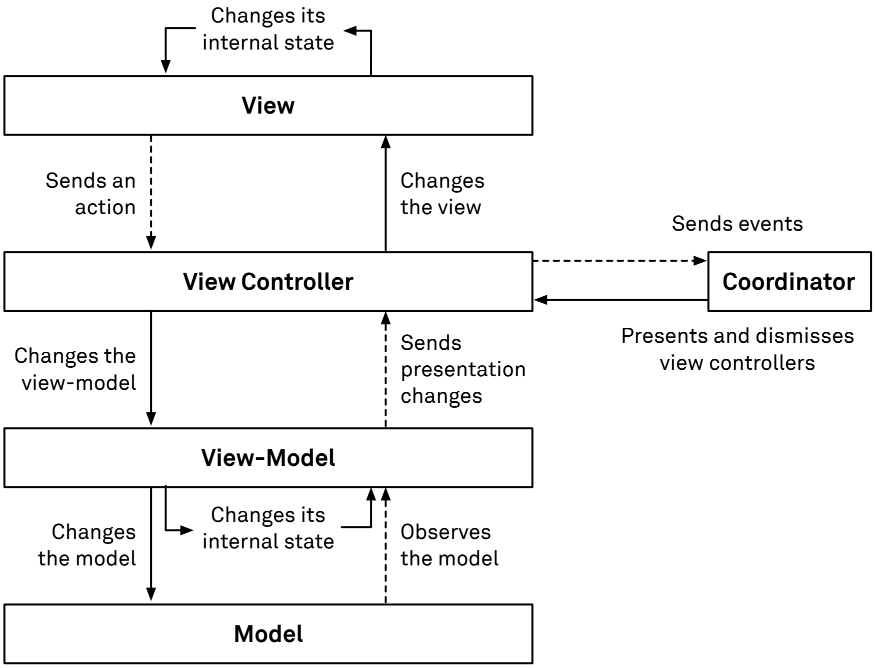
A view-model is an object that does not include compile-time references to the views or controllers. Instead, it exposes properties that describe the presentation values of the views (the values each view displays). These presentation values are derived from the underlying model objects by applying a series of transformations so that they can be directly set on the views. The actual setting of values on views is handled by bindings that take these presentation values and ensure they are set on the views whenever they change. Reactive programming is a tool for expressing this type of declarative, transformative relationship, so it's a natural fit (although not strictly necessary) for view-models. In many cases, the entire view-model can be expressed declaratively using reactive programming bindings.

Not having a reference to the view layer theoretically makes the view-model independent of the application framework and allows testing independent from the application framework.

Since the view-model is coupled to the scene, it's helpful to have an object that provides logic between scene transitions. In MVVM-C, this object is called a *coordinator*. A coordinator is an object that holds references to the model layer and understands the structure of the view controller tree so that it can provide the required model objects to the view-model for each scene.

Unlike in MVC, the view controller in MVVM-C never directly references other view controllers (or view-models, for that matter). Instead, view controllers notify the coordinator (through a delegate mechanism) about relevant view actions. The coordinator then presents new view controllers and sets their model data. In other words: the view controller hierarchy is managed by the coordinator and not by view controllers.

The resulting architecture has the following overall structure:



If we ignore the coordinator, this diagram is similar to MVC — with an additional stage between the view controller and the model. MVVM offloads most of the work that was previously in the view controller onto the view-model, but note that the view-model has no compile-time reference (solid line) in the direction of the view controller.

The view-model can be separated from the view controller and views and tested independently. Likewise, the view controller no longer has internal view state — this is moved to the view-model. The double role of the view controller in MVC (as part of the view hierarchy and also mediating interactions between the view and the model) is reduced to a single role (the view controller is solely part of the view hierarchy).

The coordinator pattern adds to this and removes yet another responsibility of view controllers: presenting other view controllers. As such, it reduces coupling between view controllers at the expense of at least one more controller-layer interface.

1. Construction

Construction of the model is unchanged from MVC and typically remains the responsibility of one of the top-level controllers. The individual model objects, however, are owned by the view-models and not by the view controllers.

The initial construction of the view hierarchy works like in MVC and is done through storyboards or in code. However, unlike in MVC, the view controller doesn't directly fetch and prepare data for each view, but instead leaves this task to the view-model. The view controller creates a view-model upon construction and then binds each view to the relevant property exposed by the view-model.

2. Updating the Model

In MVVM, the view controller receives view events in the same way as in MVC (and the connection between view and view controller is set up the same way). However, when a view event arrives, the view controller doesn't change its internal state, the view state, or the model — instead, it immediately calls a method on the view-model. In turn, the view-model changes its internal state or the model.

3. Changing the View

Unlike with MVC, the view controller doesn't observe the model. Instead, the view-model observes the model and transforms the model notifications in such a way that the view controller understands them. The view controller subscribes to the view-model's changes, typically using bindings from a reactive programming framework, but it could be any observation mechanism. When a view-model event arrives, the view controller changes the view hierarchy.

To be unidirectional, the view-model should always send model-changing view actions through the model and only notify the relevant observers after the model change has taken place.

4. View State

The view state is either in the views themselves or in the view-model. Unlike in MVC, the view controller doesn't have any view state. Changes to the view-model's view state are observed by the controller, although the controller can't distinguish between model notifications and view state change notifications. When using coordinators, the view controller hierarchy is managed by the coordinator(s).

5. Testing

Because the view-model is decoupled from the view layer and controller layer, the view-model can be tested using interface testing, rather than with MVC's integration testing. Interface tests are much simpler than integration tests, as they don't need to set up the entire hierarchy of components.

To cover as much as possible using interface tests, the view controller needs to be as simple as possible, but the parts that aren't moved out of the view controller still need to be tested separately. In our implementation, this includes both interaction with the coordinator and the initial construction.

Importance of Model-View-ViewModel

MVVM is the most popular application design pattern on iOS that is not a direct variant of MVC. That said, it's not dramatically different from MVC either; both are structured around view controller scenes and use most of the same machinery.

The biggest difference is probably the use of reactive programming to express logic in the view-model as a series of transformations and dependencies. Using reactive programming to clearly describe the relationship between model object and presentation values provides an important lesson in understanding dependencies in applications more generally.

Coordinators are a useful pattern in iOS because managing the view controller hierarchy is such an important concept. Coordinators are not inherently tied to MVVM, and they can be used with MVC or other patterns as well.

History

MVVM was formulated by Ken Cooper and Ted Peters, who were working at Microsoft on what would become the Windows Presentation Foundation (WPF), an application framework for Microsoft .NET released in 2005.

WPF uses a declarative XML-based language called XAML to describe the properties on the view-model to which the view binds. In Cocoa, without XAML, frameworks like RxSwift, along with code (usually in the controller), must be used to perform the binding between view-model and the view.

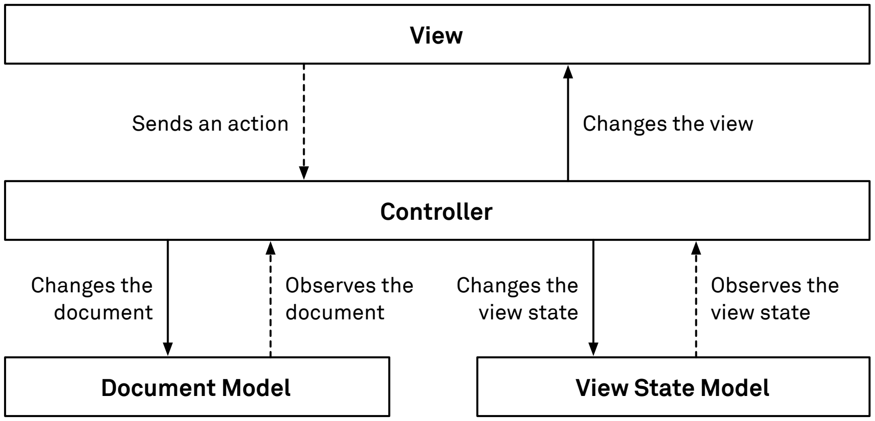
MVVM is very similar to the MVP pattern mentioned in the history of MVC. However, as Cooper and Peters described it, MVVM requires explicit framework support for the binding between the view and the view-model, whereas presenters traditionally propagate changes manually between the two.

Coordinators in iOS were recently (re)popularized by Soroush Khanlou, who described them on his website in 2015, though they are based on the older pattern of application controllers that have been in Cocoa and other platforms for decades.

Model-View-Controller+ViewState

MVC+VS is an attempt to bring a unidirectional data flow approach to otherwise standard MVC. Rather than following the two or three different paths view state can take in standard Cocoa MVC, it aims to make the handling of view state more manageable. In MVC+VS, we explicitly identify and represent all view state in a new model object called the view state model.

In MVC+VS, we don't ignore any navigation change, row selection, text field edit, toggle, modal presentation, or scroll position change (or other view state changes). Instead, we send these changes to a view state model action. Each view controller observes the view state model, which makes communicating changes straightforward. In presentation or interaction logic, we never read view state from the views, but instead read it from the view state model:



The resulting diagram is similar to MVC, but instead of the controller's internal feedback loop (which was used for the view state), there is now a separate view state loop (similar to the model loop).

1. Construction

While it's still the responsibility of the view controllers to apply document model data to the views — as in typical MVC — the view controller also applies and subscribes to the view state. This forces a larger amount of work through the notification observing functions — and since there's both a view state model and a document model to observe, there are more observing functions than in typical MVC.

2. Updating the Model

When a view action happens, the view controller changes either the document model (unchanged from MVC) or the view state model. We don't change the view hierarchy directly; instead, all changes flow through the document model and view state model.

3. Changing the View

The controller observes both the document model and the view state model and updates the view hierarchy only when changes occur.

4. View State

View state is made explicit and is extracted from the view controller. The treatment is identical to the model: the controllers observe the view state model and change the view hierarchy accordingly.

5. Testing

In MVC+VS, we use integration testing like in MVC, but the tests themselves are quite different. Every test starts with an empty root view controller, and by setting the document model and view state model, the root view controller builds up the entire hierarchy of views and view controllers. The hardest part of integration testing in MVC (setting up all the components) is done automatically in MVC+VS. To test a different view state, we can set the global view state and all view controllers will adjust themselves.

Once the view hierarchy is constructed, we write two kinds of tests. The first kind tests whether or not the view hierarchy is constructed according to our expectations, and the second kind tests whether or not view actions change the view state correctly.

Importance of Model-View-Controller+ViewState

MVC+VS is primarily a teaching tool for the concept of view state.

In an application that is otherwise standard MVC, adding a view state model and observing it in each view controller (in addition to the model observing already present) offers several advantages: arbitrary state restoration (not reliant on storyboards or `UIStateRestoration`), full user interface state logging, and the ability to jump between different view states for debugging purposes.

History

This specific formulation was developed by Matt Gallagher in 2017 as a teaching tool for the concepts of unidirectional data flow and time travel in user interfaces. The goal was to make the minimum number of changes required to a traditional Cocoa MVC app so that a snapshot could be taken of the state of the views on every action.

ModelAdapter-ViewBinder

MAVB is an experimental pattern centered around bindings. In this pattern, there are three important concepts: view binders, model adapters, and bindings.

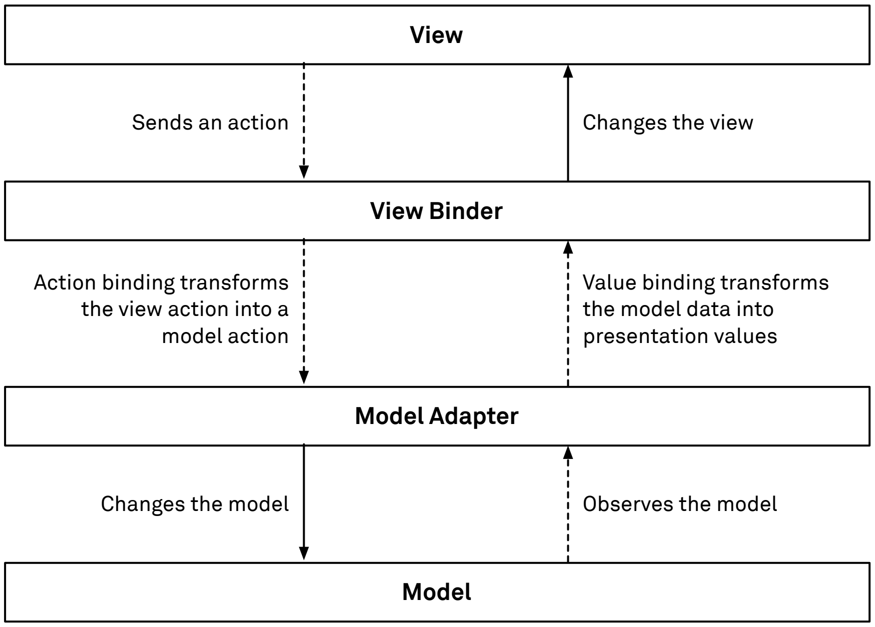
A view binder is a wrapper class around a view class (or view controller class): it constructs the view and exposes a list of bindings for it. Some bindings provide data to the view (e.g. a label's text), while others emit events from the view (e.g. a button click or a navigation change).

Although view binders can contain dynamic bindings, view binders themselves are immutable. This makes MAVB another declarative pattern: you declare your view binders and their behavior once rather than changing view binders over time.

A model adapter is a wrapper around mutable state and is implemented using a so-called *reducer*. The model adapter provides an input binding (to send events into) and an output binding (to receive updates).

In MAVB, you don't create views directly; rather, you only construct view binders. Likewise, you never work with mutable state, except in model adapters. The transformation between view binders and model adapters (in both directions) is done by transforming the bindings (using standard reactive programming techniques).

MAVB removes the need for a controller layer. The construction logic is expressed through view binders, the transformation logic is expressed in the bindings, and the state changes are expressed through model adapters. The resulting diagram looks like this:


View
Sends an action
Changes the view
View Binder
Action binding transforms the view action into a model action
Value binding transforms the model data into presentation values
Model Adapter
Changes the model
Observes the model
Model

1. Construction

A model adapter (wrapping the main model) and a view state adapter (wrapping the top-level view state) are typically constructed in the main.swift file before any views.

View binders are constructed using plain functions. These functions take the necessary model adapters as parameters. The actual Cocoa view objects are constructed by the framework.

2. Updating the Model

When a view (or view controller) can emit actions, the corresponding view binding allows us to specify an action binding. Here, the data flows from the view to the output end of the action binding. Typically, the output end is connected to a model adapter, and the binding is used to transform the view event into a message that the model adapter can understand. This message is then used by the model adapter's reducer to change the state.

3. Changing the View

After a model adapter's state changes, it emits notifications through its output signal. In a view binder, we can transform the model adapter's output signal and bind it to a view property. Therefore, the view property automatically changes when a notification is emitted.

4. View State

View state is considered a part of the model layer. View state actions and view state notifications take the same path as model actions and model notifications.

5. Testing

In MAVB, we test our code by testing the view binders. As a view binder is a list of bindings, we can verify that the bindings contain the items we expect and that they are configured correctly. We can use the bindings to test both initial construction and changes.

Testing in MAVB is similar to testing in MVVM. However, in MVVM, it's possible to have logic in the view controller, which potentially leaves untested code between the view-model and view. As MAVB doesn't have view controllers, and the binding is the only piece of code between the model adapter and view binder, it's much easier to guarantee full test coverage.

Importance of ModelAdapter–ViewBinder

Of the major patterns we're examining, MAVB has no direct precedent — it is not an implementation of a pattern from another platform or a variation on another pattern. It is its own thing: experimental and a little weird. Its inclusion here is to show something that's proudly different. But that's not to say it takes no lessons or ideas from other patterns — bindings, reactive programming, domain-specific languages, and reducers are all well-known ideas.

History

MAVB was first described by Matt Gallagher on the Cocoa with Love website. The pattern draws inspiration from Cocoa bindings, Functional Reactive Animation, ComponentKit, XAML, Redux, and experiences working in Cocoa view controllers with thousands of lines.

The implementation in this book uses the CwlViews framework to handle the view construction, binder, and adapter implementations.

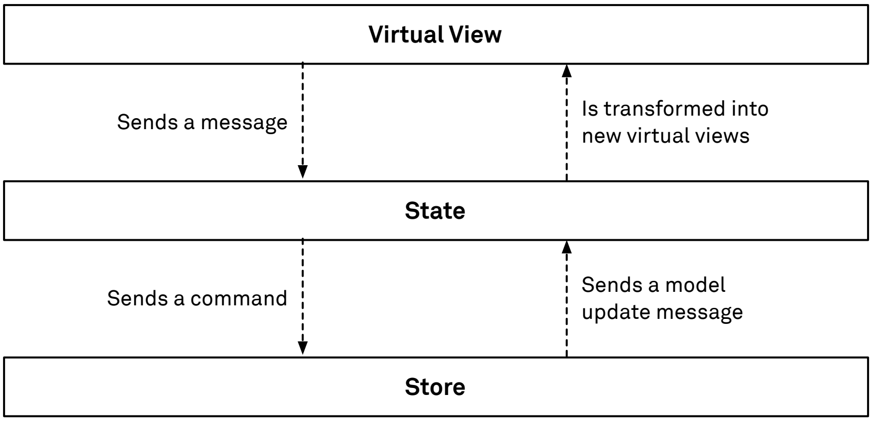
The Elm Architecture

TEA is a more radical departure from MVC. In TEA, the model and all view state are integrated as a single state object, and all changes in the application occur by sending *messages* to this state object, which processes the messages in a state-updating function called a *reducer*.

In TEA, each change to the state creates a new *virtual view hierarchy*, which is composed of lightweight structs that describe how the view hierarchy should look. The virtual view hierarchy allows us to write our views as a *pure function*; the virtual view hierarchy is always directly computed from the state, without any side effects. When the state changes, we recompute the virtual view hierarchy using the same function instead of changing the view hierarchy directly.

The `Driver` (which is part of the TEA framework and holds references to the other layers in TEA) compares the virtual view hierarchy to the `UIView` hierarchy and applies the necessary changes to make the views match their virtual counterparts. The driver is a component of the TEA framework that we instantiate once for our application, and it doesn't know about the specifics of the application itself. Instead, we pass this information into its initializer: the app's initial state, a function to update the state from a message, a function to render the virtual view hierarchy for a given state, and a function to compute the subscriptions for a given state (for example, we can subscribe to the store's change notifications).

From the perspective of a user of the framework, a block diagram of changes in TEA looks like this:


Virtual View
Sends a message
Is transformed into new virtual views
State
Sends a command
Sends a model update message
Store

If we trace around the upper two layers of this diagram, we get the feedback loop between the view and the model that we showed at the start of this chapter; it's a loop from view to state and then back to the view (mediated through the TEA framework).

The lower loop represents TEA's way of dealing with side effects (like writing data to disk): when processing a message in the state's update method, we can return a command, and these commands are executed by the driver. In our case, the most important commands are for changing the contents of the store. The store in turn is observed by subscriptions owned by the driver. These subscriptions can trigger messages to change the state, which triggers view rerendering in response.

The structure of these event loops makes TEA another example of a design pattern following the unidirectional data flow principle.

1. Construction

The state is constructed on startup and passed to the runtime system (the driver). The runtime system owns the state. The store is a singleton.

The initial view hierarchy is constructed through the same path that's used for updates later on: a virtual view hierarchy is computed from the current state, and the runtime system takes care of updating the real view hierarchy to match the virtual view hierarchy.

2. Updating the Model

Virtual views have messages associated with them that get sent when a view event occurs. The driver receives these messages and uses the update method to change the state. The update method can return a command (a side effect) — for example, a change we want to make in the store. The driver interprets this command and executes it. The TEA framework makes it impossible for the views to change the state or store directly.

3. Changing the View

The runtime system takes care of this. The only way to change the views is by changing the state. Therefore, there is no difference between initial construction and updating the view hierarchy.

4. View State

The view state is included in the state. As the view is computed directly from the state, the navigation and interaction state are automatically updated as well.

5. Testing

In most architectures, it takes a lot of effort to test that components are connected. In TEA, we don't need to test this, as the driver automatically takes care of it. Similarly, we don't need to test that the view changes correctly when the state changes. Instead, we only need to test that the correct virtual view hierarchy is computed for a given state.

To test changes to the state, we can take a given state and use the `update` function with a message to change the state. We can then compare the before and after states and verify that `update` returns the expected commands for a given state and message. In TEA, we can also test that the correct subscriptions are returned for a given state. Like the view hierarchy, both the `update` function and the subscriptions are pure functions.

Because all the components (computing the virtual view hierarchy, the update function, and the subscriptions) are pure functions, we can test them in complete isolation. We don't need to instantiate any framework components: instead, we pass in parameters and verify the result. Most tests for our TEA implementation are straightforward.

Importance of The Elm Architecture

TEA was first implemented in Elm, a functional programming language. TEA is therefore a look into how GUI programming can be expressed in a functional way. TEA is also the oldest unidirectional data flow architecture.

History

The Elm language is a functional programming language initially designed by Evan Czaplicki for building frontend web apps. TEA is a pattern attributed to the Elm community, and it reportedly emerged naturally from the constraints of the language and the target environment. The ideas behind it have influenced other web-based frameworks like React, Redux, and Flux. There is no authoritative implementation of TEA in Swift, but there are a number of research projects. For this book, we have built our own interpretation of the pattern in Swift. The main work was done by Chris Eidhof in 2017. While our specific implementation is not “production ready,” many of the ideas can be used in production code.

Throughout the book, we use the word *state*; in Elm, this is called *model*. We chose to call it *state* even when referring to TEA, as the word model has a different meaning in this book.

Networking

There is also a chapter about networking in this book. In it, we take the MVC version of our app and show two different ways of adding networking. In the first approach, *controller-owned networking*, we replace the model layer with a web service. In the second approach, *model-owned networking*, we add networking on top of the model layer.

Controller-owned networking is simpler to start with: each view controller performs requests as needed and caches the results locally. However, this pattern becomes impractical once the data needs to be shared across view controllers. At this point, it is often easier to move to model-owned networking. With networking being an extension of the model layer, we have an established mechanism for sharing data and communicating changes.

Patterns Not Covered

We've chosen to cover five different patterns in this book. Three of them are MVC variants, and two of them are experimental or not production ready. Why these patterns and not some of the other design patterns that exist in shipping applications? Below, we'll briefly outline some of the other contenders and why we didn't include them.

Model-View-Presenter

Model-View-Presenter (MVP) is a pattern that's popular on Android and has implementations on iOS. It sits roughly between standard MVC and MVVM in terms of overall structure and technologies used.

MVP uses a separate presenter object that occupies the same position the view-model occupies in MVVM. Relative to the view-model, the presenter omits reactive programming, along with the expectation that presentation values are exposed as properties on the interface. However, when values need to change, the presenter immediately pushes them down to the views (which are exposed to the presenter as a protocol).

In terms of abstraction, MVP is similar to MVC. Cocoa MVC, despite its name, *is* MVP — it is derived from Taligent's original MVP implementation in the 1990s. Views, state, and related logic are represented identically in both patterns. The difference is that modern MVP has a separate presenter entity, and it uses protocol boundaries between the presenter and view controller, whereas Cocoa MVC lets the controller directly reference the view.

Some developers believe the protocol separation is necessary for testing. When we discuss testing, we'll show how standard MVC can be fully tested without any separation, and as such, we feel that MVP isn't different enough. If we strongly desire a fully decoupled representation for testing, we think MVVM's approach is simpler: have the view controller *pull* the values from the view-model via observing rather than having the presenter push values to a protocol.

VIPER, Riblets, and Other “Clean” Patterns

VIPER, Riblets, and similar patterns are efforts to bring Robert Martin’s “Clean Architecture” from web applications to iOS development by spreading the roles of the controller across three to four different classes with a strict sequential ordering. Each class is forbidden from directly referencing preceding classes in the sequence.

To enforce the rules about referencing in one direction only, these patterns require a *lot* of protocols, classes, and passing of data between layers. For this reason, many developers using these patterns use code generators. Our feeling is that code generators — and any patterns verbose enough to imply their need — are misguided. Attempts to bring “Clean Architecture” to Cocoa usually claim to manage “massive view controllers,” but ironically, do so by making the code base even larger.

While interface decomposition is a valid approach for managing code size, we feel it should be performed as needed, rather than methodically and per view controller. Decomposition should be performed along with knowledge of the data and tasks involved so that the best abstraction — and hence the best reduction in complexity — can be achieved.

Component-Based Architecture (React Native)

If you prefer programming in JavaScript to Swift, or if your application relies heavily on web API interactions that work better in JavaScript, you might consider React Native. However, this book is focused on Swift and Cocoa, so we’re limiting the patterns explored to those domains.

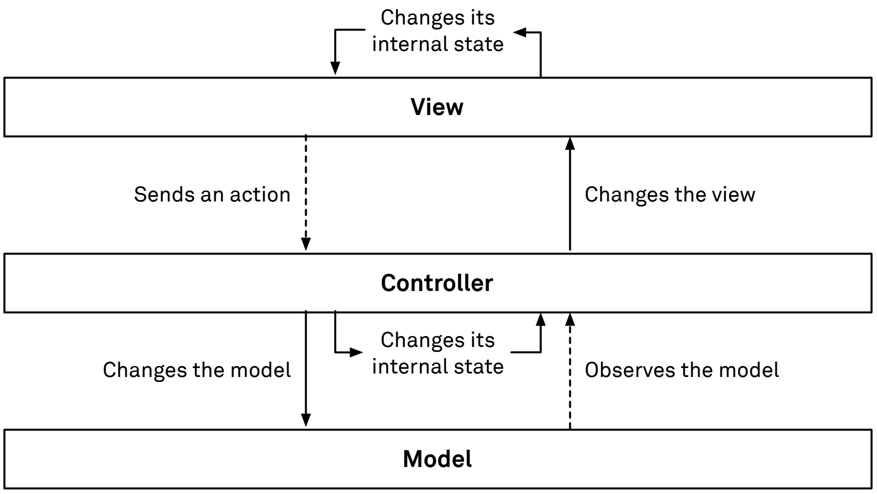
If you’re looking for something like React Native but in Swift, then take a look at our exploration of TEA. The MAVB implementation also takes inspiration from ComponentKit — itself inspired by React — and uses a DSL-like syntax for declarative, transformational view construction with parallels to the `render` function in React `Components`.

Model-View-Controller

3

The Model-View-Controller (MVC) pattern is the baseline for all other patterns discussed in this book. It is also the authoritative pattern of the Cocoa frameworks and the oldest pattern we will discuss.

The key premise of MVC is that the model layer and view layers are brought together by the controller layer, which constructs and configures the other two layers and mediates communication — in both directions — between model and view objects. Thus, the controller layer represents the backbone upon which the application feedback loop is formed in MVC apps:



MVC is based on classic object-oriented principles: objects manage their behaviors and state internally and communicate via class and protocol interfaces; view objects are typically self-contained, reusable objects; and model objects are presentation independent and avoid dependence upon the rest of the program. It is therefore the responsibility of the controller layer to combine the other two layers into a program.

Apple describes MVC as a collection of three distinct subpatterns:

1. Composite pattern — views are assembled together into hierarchies, and sections of the hierarchy are managed in groups by controller objects.

2. Strategy pattern — controller objects are mediators between the view and model and manage all application-specific behaviors for reusable, application-independent views.
3. Observer pattern — objects dependent on model data must subscribe to receive updates.

The MVC pattern is often interpreted quite loosely such that these subpatterns — in particular, the observer pattern — are not always followed. However, we will include all three as part of our implementation.

The MVC pattern has some well-known deficiencies, the foremost of which is the *massive view controller* problem (which uses the same MVC acronym as the pattern), wherein the controller layer takes on too many responsibilities. MVC also presents difficulties in testing, thereby making unit tests and interface tests difficult or impossible. Despite these shortcomings, MVC remains the simplest pattern to use for iOS apps, can scale to suit any program — provided you understand its shortcomings and how to counter them — and can include robust testing.

In the next section, we will discuss the implementation details of the MVC variant of the Recordings app. We only show parts of the source code in the book, but the complete source code (of all variants of the app) is available on GitHub.

Exploring the Implementation

Construction

The construction process in Cocoa MVC largely follows the default startup process provided by the Cocoa framework. The key aim of this process is to ensure the construction of three objects: the `UIApplication` object, the application delegate, and the root view controller of the main window. The configuration of this process is distributed across three files, which are by default named `Info.plist`, `AppDelegate.swift`, and `Main.storyboard`.

These three objects — all part of the controller layer — provide configurable locations to continue the startup process in, thereby placing the controller layer in charge of all construction.

Connecting Views to Initial Data

Views in an MVC app do not directly reference model objects; they are kept independent and reusable. Instead, model objects are stored in the view controller. This makes the view controller a non-reusable class, but that's the purpose of view controllers: to give application-specific knowledge to other components in the program.

The model object stored in the view controller gives the latter *identity* (letting the view controller understand its position in the program and how to talk to the model layer). The view controller extracts and transforms the relevant property values from the model object before setting these transformed values on any child views.

Exactly how this identity object is set on a view controller varies. In the Recordings app, there are two different strategies used for setting the initial model value on view controllers:

1. Immediately accessing a global model object for the controller based on the controller's type or location in the controller hierarchy.
2. Initially setting references to model objects to `nil` and keeping everything in a blank state until a non-`nil` value is provided by another controller.

A third strategy — passing the model object on construction (also known as dependency injection) — would be preferred if it were possible. However, the storyboard construction process typically prevents passing construction parameters to view controllers.

The `FolderViewController` is an example of the first strategy. Each folder view controller initially sets its `folder` property as follows:

```
var folder: Folder = Store.shared.rootFolder {
   // ...
}
```

This means that on initial construction, every folder view controller assumes it represents the root folder. If it actually is a child folder view controller, then the parent folder view controller will set the folder value to something else during `perform(segue:)`. This approach ensures that the `folder` property in the folder view controller is not optional, and that the code can be written without having to conditionally test for the existence of the folder object, since there is always at least the root folder.

In the Recordings app, the model object Store.shared is a lazily constructed singleton. This means that the moment when the first folder view controller attempts to access the Store.shared.rootFolder is probably the point when the shared store instance is constructed.

The PlayViewController (the detail view in the top-level split view) uses an initially nil optional reference for its model object and therefore represents the second strategy for setting the identity reference:

```
var recording: Recording? {
   // ...
}
```

When the value of recording is nil, the play view controller shows a blank ("No recording selected") display. Note that nil is an expected state, and it is not used as a workaround for Swift's strict initialization requirements. The recording is set from the outside, either from the folder view controller or by the state restoration process. In either case, once this primary model object is set, the controller responds by updating the views.

Another example of the controller responding to a primary model change is in the folder view controller. The navigation title (the display of the folder name in the navigation bar at the top of the screen) must be updated when the folder is set:

```
var folder: Folder = Store.shared.rootFolder {
   didSet {
      tableView.reloadData()
      if folder === folder.store?.rootFolder {
         title = .recordings
      } else {
         title = folder.name
      }
   }
}
```

The call to tableView.reloadData() will ensure that all the UITableViewCells displayed are updated too.

As a rule, whenever we read the initial model data, we must also observe changes to the model. In the folder view controller's viewDidLoad, we add the view controller as an observer of model notifications:

```
override func viewDidLoad() {
  super.viewDidLoad()
  // ...
  NotificationCenter.default.addObserver(self,
    selector: #selector(handleChangeNotification(_:)),
    name: Store.changedNotification, object: nil)
}
```

We'll discuss how we handle these notifications in the handleChangeNotification method covered later on in this chapter.

State Restoration

State restoration in MVC uses the storyboard system, which acts as part of the controller layer. Opting into this system requires the following methods to be implemented on the AppDelegate:

```
func application(_ application: UIApplication,
  shouldSaveApplicationState coder: NSCoder) -> Bool
{
  return true
}

func application(_ application: UIApplication,
  shouldRestoreApplicationState coder: NSCoder) -> Bool
{
  return true
}
```

Once these two methods are implemented, the storyboard system takes over. View controllers that should be saved and restored automatically by the storyboard system are configured with a restoration identifier. For example, the root view controller in the Recordings app has the restoration identifier splitController specified on the Identity inspector in the storyboard editor. Similar identifiers exist for every scene in the Recordings app except the RecordViewController (which is deliberately non-persistent).

While the storyboard system preserves the existence of these view controllers, it will not preserve the model data stored in each view controller without us doing further work. To store this extra state, each view controller must implement encodeRestorableState(with:) and decodeRestorableState(with:). This is the implementation on the FolderViewController:

```
override func encodeRestorableState(with coder: NSCoder) {
   super.encodeRestorableState(with: coder)
   coder.encode(folder.uuidPath, forKey: .uuidPathKey)
}
```

The encoding part is simple — the FolderViewController saves the uuidPath that identifies its Folder model object. The decoding part is a little more complicated:

```
override func decodeRestorableState(with coder: NSCoder) {
   super.decodeRestorableState(with: coder)
   if let uuidPath = coder.decodeObject(forKey: .uuidPathKey) as? [UUID],
      let folder = Store.shared.item(atUUIDPath: uuidPath) as? Folder
   {
      self.folder = folder
   } else {
      if let index = navigationController?.viewControllers.index(of: self),
         index != 0
      {
         navigationController?.viewControllers.remove(at: index)
      }
   }
}
```

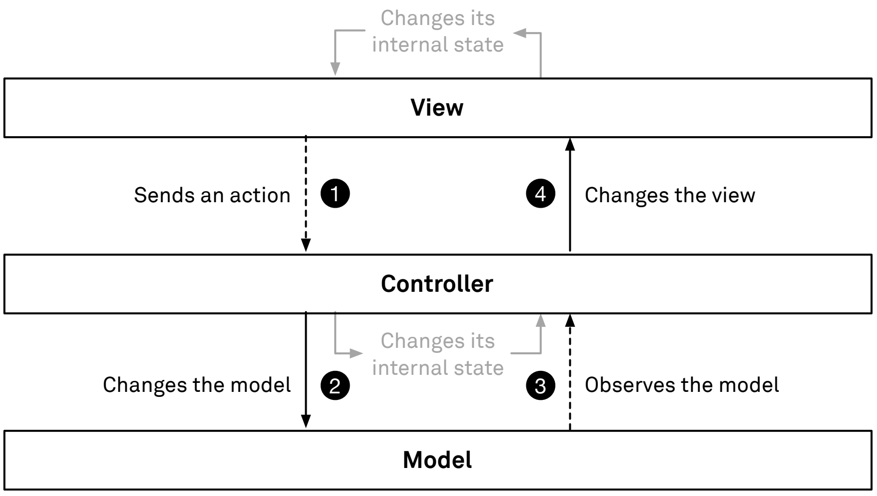
After decoding the uuidPath, the FolderViewController must check that the item still exists in the store so that it can set its folder property to the item. If the item does not exist in the store, then the FolderViewController must attempt to remove itself from its parent navigation controller's list of view controllers.

Changing the Model

The broadest interpretations of MVC don't include any details about how to implement the model, how model changes should occur, or how the view should respond to the changes. In the earliest versions of macOS, following the precedent set by the earlier Document-View patterns, it was common to have controller objects like `NSWindowController` or `NSDocument` directly change the model in response to view actions and then directly update the view as part of the same function.

For our implementation of MVC, we believe actions that update the model should not occur in the same function as changes to the view hierarchy. Instead, these actions must return without assuming any effect on the model state. After the construction phase, changes to the view hierarchy may occur only as a result of observation callbacks — following the observer pattern that is part of the MVC formulation.

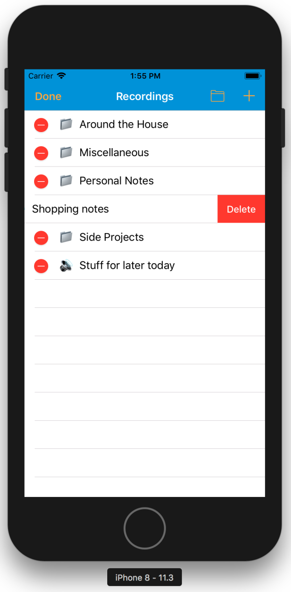
The observer pattern is essential to maintaining a clean separation of model and view in MVC. The advantage of this approach is that we can be sure the UI is in sync with the model data, no matter where the change originated from (e.g. from a view event, a background task, or the network). Additionally, the model has a chance to reject or modify the requested change:



In this next section, we'll look at the steps involved in deleting an item from a folder.

Step 1: Table View Sends Action

In our example app, the data source of the table view is set to the folder view controller by the storyboard. To handle the tap on the delete button, the table view invokes tableView(_:commit:forRowAt:) on its data source:



Step 2: View Controller Changes Model

The implementation of tableView(_:commit:forRowAt:) looks up the item that should be deleted (based on the index path) and asks the parent folder to remove it:

```
override func tableView(_ tableView: UITableView,
   commit editingStyle: UITableViewCellEditingStyle,
   forRowAt indexPath: IndexPath)
{
   folder.remove(folder.contents[indexPath.row])
}
```

Note that we are not deleting the cell from the table view directly. This only happens once we observe the model change.

The remove method on Folder notifies the item that it has been deleted by calling item.deleted(). It then removes it from the folder's contents. Then it tells the store to persist the data, including detailed information about the change it just made:

```
func remove(_ item: Item) {
  guard let index = contents.index(where: { $0 === item }) else { return }
  item.deleted()
  contents.remove(at: index)
  store?.save(item, userInfo: [
    Item.changeReasonKey: Item.removed,
    Item.oldValueKey: index,
    Item.parentFolderKey: self
  ])
}
```

If the deleted item is a recording, the associated file is removed from the file system by the call to item.deleted(). If it is a folder, it recursively removes all contained subfolders and recordings.

Persisting the model objects and sending the change notification occurs in the call to the store's save method:

```
func save(_ notifying: Item, userInfo: [AnyHashable: Any]) {
  if let url = baseURL, let data = try? JSONEncoder().encode(rootFolder) {
    try! data.write(to: url.appendingPathComponent(.storeLocation))
    // error handling ommitted
  }
  NotificationCenter.default.post(name: Store.changedNotification,
    object: notifying, userInfo: userInfo)
}
```

Step 3: View Controller Observes Model Changes

In the construction section, we saw that the folder view controller sets up an observation of the store's change notification in viewDidLoad:

```
override func viewDidLoad() {
   super.viewDidLoad()
   // ...
   NotificationCenter.default.addObserver(self,
      selector: #selector(handleChangeNotification(_:)),
      name: Store.changedNotification, object: nil)
}
```

The call to save on the store from the previous step sends the change notification. In response, this observation triggers and calls handleChangeNotification.

Step 4: View Controller Changes View

When the store change notification arrives, the view controller's handleChangeNotification method interprets it and makes the corresponding change in the view hierarchy.

A minimalist handling of notifications might involve reloading the table data whenever any type of model notification arrives. In general though, the correct handling of model notifications involves properly understanding the data change described by the notification. Accordingly, our implementation communicates the nature of the model change — including the specific row that has changed and the type of change that has occurred — through the notification's userInfo dictionary.

In this example, handling the notification involves a call to tableView.deleteRows(at:with:):

```
@objc func handleChangeNotification(_ notification: Notification) {
   // ...
   if let changeReason = userInfo[Item.changeReasonKey] as? String {
      let oldValue = userInfo[Item.newValueKey]
      let newValue = userInfo[Item.oldValueKey]
      switch (changeReason, newValue, oldValue) {
      case let (Item.removed, _, (oldIndex as Int)?):
         tableView.deleteRows(at: [IndexPath(row: oldIndex, section: 0)],
            with: .right)
      // ...
      }
   } else {
      tableView.reloadData()
   }
}
```

Noticeably absent from this code: we have not updated any data on the view controller itself. The folder value on the view controller is a shared reference directly to the object in the model layer, so it is already up to date. After the above call to tableView.deleteRows(at:with:), the table view will call the data source implementations on the folder view controller and they will return the latest state of the data accessed through the folder shared reference. In the MVC+VS chapter, we will look at an alternative model layer with value types instead of objects.

We should also clarify that this notification handling relies on a shortcut: the model stores items in the same lexically sorted order used for display. This is *not* ideal; the model should not really know *how* its data will be displayed. A conceptually cleaner implementation (that requires more work) would involve the model emitting *set* mutation information (not array mutation information) and the notification handler using its own sorting, combined with before and after states, to determine deleted indices. In the chapter on MAVB, we keep sorting out of the model layer and move it into the view binders.

This completes the "changing the model" event loop in MVC. Since we updated the UI only in response to the change in the model (and not directly in response to the view action), the UI will update correctly, even if the folder is removed from the model for other reasons (for example, a network event), or if the change is rejected by the model. This is a robust approach to ensure the view layer does not get out of sync with the model layer.

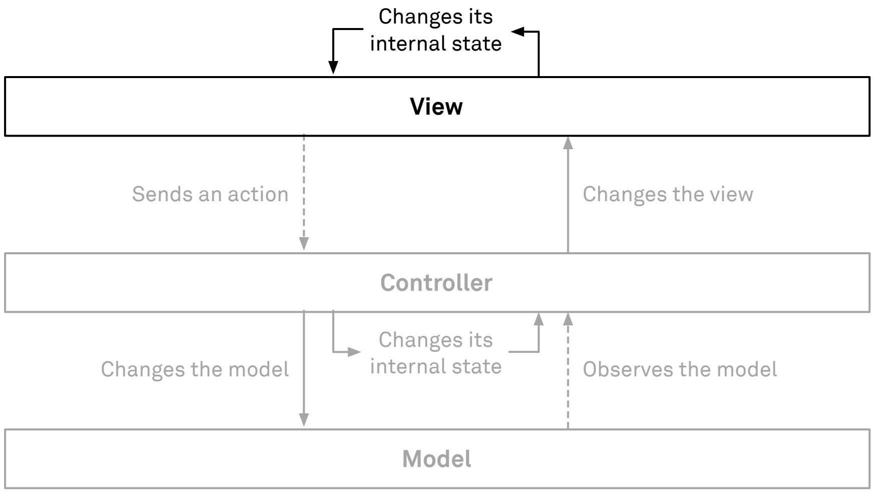
Changing the View State

The model layer in MVC has its origins in typical document-based applications: any state that is written to the document during a save operation is considered part of the model. Any other state — including navigation state, temporary search and sort values, feedback from asynchronous tasks, and uncommitted edits — is traditionally excluded from the definition of the model in MVC.

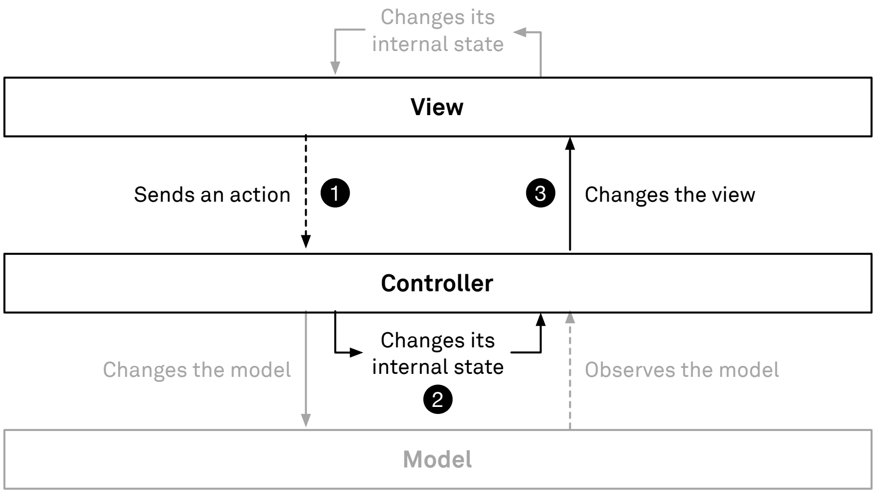
In MVC, this "other" state — which we collectively call view state — does not have a description in the pattern. In accordance with traditional object-oriented principles, any object may have internal state and the object is not required to communicate changes in that internal state to the remainder of the program.

Due to this internal treatment, view state does not necessarily follow any clear path through the program. Any view or controller may include state, which it updates in response to view actions. View state is handled as locally as possible: a view or view controller can autonomously update its view state in response to a user event.

Most UIViews have internal state that they may update in response to view actions without propagating the change further. For example, a UISwitch may change from ON to OFF in response to a user touch event, as follows:



The event loop becomes one step longer if the view cannot change its state itself — for example, when the label of a button should change after it's tapped (as with our play button) or when a new view controller is pushed onto the navigation stack in response to the user tapping a table view cell.



The view state is still contained within one particular view controller and its views. However, compared to a view changing its own state, we now get the chance to customize the state change of a view (as with the play button title in the first example below) or to make view state changes across views (exemplified by the second example below of pushing a new folder view controller).

Example 1: Updating the Play Button

The play button in the play view controller changes its title from "Play" to "Pause" to "Resume" depending on the play state. From the user's perspective, when the button says "Play," tapping it changes it to "Pause"; tapping it again before playback ends changes it to "Resume." Let's take a closer look at the steps Cocoa MVC takes to achieve this.

Step 1: The Button Sends an Action to the View Controller

The play button connects to the play view controller's play method using an IBAction in the storyboard. Tapping the button calls the play method:

```
@IBAction func play() {
  // ...
}
```

Step 2: The View Controller Changes Its Internal State

The first line in the play method updates the state of the audio player:

```
@IBAction func play() {
   audioPlayer?.togglePlay()
   updatePlayButton()
}
```

Step 3: The View Controller Updates the Button

The second line in the play method calls updatePlayButton, which directly sets the new title of the play button, depending on the state of the audio player:

```
func updatePlayButton() {
   if audioPlayer?.isPlaying == true {
      playButton?.setTitle(.pause, for: .normal)
   } else if audioPlayer?.isPaused == true {
      playButton?.setTitle(.resume, for: .normal)
   } else {
      playButton?.setTitle(.play, for: .normal)
   }
}
```

.pause, .resume, and .play are localized strings defined as static constants on String.

At this point, we're already done, and the smallest number of components necessary were involved in the process: the button sends the event to the play view controller, and in turn, the play view controller sets the new title.

Example 2: Pushing a Folder View Controller

The folder view controller's table view shows two kinds of items: recordings and subfolders. When the user taps on a subfolder, a new folder view controller is configured and pushed onto the navigation stack. Since we use a storyboard with segues to achieve this, the concrete steps below only loosely relate to the steps in the diagram above. However, the general principle stays the same.

Step 1: Triggering the Segue

Tapping a subfolder cell triggers a showFolder segue, as the cell is connected to the folder view controller via a push segue in the storyboard. This causes UIKit to create a new folder view controller instance for us.

This step is a variant of the target/action pattern. There's more UIKit magic involved behind the scenes, but the result is that the prepare(for:sender:) method on the originating view controller gets called.

Steps 2 & 3: Configuring the New Folder View Controller

The current folder view controller gets notified by prepare(for:sender:) about the segue that is about to occur. After we check the segue identifier, we configure the new folder view controller:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  guard let identifier = segue.identifier else { return }
  if identifier == .showFolder {
    guard
      let folderVC = segue.destination as? FolderViewController,
      let selectedFolder = selectedItem as? Folder
    else { fatalError() }
    folderVC.folder = selectedFolder
  }
  // ...
}
```

First, we check that the destination view controller has the right type and that we have selected a folder. If either of these conditions doesn't hold, it is a programming error and we crash. If everything is present as expected, we set the subfolder on the new folder view controller.

The storyboard machinery takes over the actual presentation of the new view controller. After we configure the new view controller, UIKit pushes it onto the navigation stack (we don't need to call `pushViewController` ourselves). Pushing the view controller onto the navigation stack causes the navigation controller to install the view controller's view in the view hierarchy.

Similar to the play button example, the UI event (selecting the subfolder cell) is handled as locally as possible. The segue mechanism obscures the exact path of the event, but nevertheless, from the perspective of our code, no other component but the originating view controller is involved. The view state is implicitly represented by the involved views and view controllers.

To share view state between different parts of the view hierarchy, we must find the common ancestor in the view (controller) hierarchy and manage the state there. For example, to share the play state between the play button label and the player, we store the state in the play view controller. When we need a view state that's used by almost all components (for example, a Boolean value that captures whether or not our application is in dark mode), we have to put it in one of the topmost controller objects (e.g. the application delegate). However, in practice, it's uncommon to put this view state in a controller object at the top of the hierarchy, because that would require a communication channel between each layer of the hierarchy. As such, most people opt to use a singleton instead.

Testing

Automated tests come in a number of different forms. From smallest to largest granularity, these include:

- → Unit tests (isolating individual functions and testing their behavior).
- → Interface tests (using interface inputs — usually functions — and testing the result via interface outputs — usually different functions).
- → Integration tests (testing the program — or significant subsections — as a whole).

While interface and data-driven regression tests have been around for 40 years or more, modern unit testing didn't really start until the 1990s, and it took at least another decade for unit testing to become common in applications. Even now, many apps have no regular testing outside of human-driven tests.

It should not be surprising then that the Cocoa MVC pattern, which is more than 20 years old, was created without unit testing in mind. We can test the model however we choose — since it is independent of the rest of the program — but that won't help us test user-facing state. We can use Xcode's UI tests — automated scripts that run our entire program and attempt to read the screen using the VoiceOver/Accessibility API — but these are slow and prone to timing issues, and it's difficult to extract precise results.

If we want to use exact code-level testing of the controller and view layers in MVC, the only option available is writing integration tests. Integration tests involve constructing a self-contained version of the app, manipulating part of it, and reading from other parts to ensure results are propagated between objects as expected.

For the Recordings app, the tests require a store, so we construct one without a URL (in-memory only) and add some test items (folders and recordings):

```
func constructTestingStore() -> Store {
   let store = Store(url: nil)

   let folder1 = Folder(name: "Child 1", uuid: uuid1)
   let folder2 = Folder(name: "Child 2", uuid: uuid2)
   store.rootFolder.add(folder1)
   folder1.add(folder2)

   let recording1 = Recording(name: "Recording 1", uuid: uuid3)
   let recording2 = Recording(name: "Recording 2", uuid: uuid4)
   store.rootFolder.add(recording1)
   folder1.add(recording2)

   store.placeholder = Bundle(for: FolderViewControllerTests.self)
      .url(forResource: "empty", withExtension: "m4a")!

   return store
}
```

The store.placeholder is a store feature used for testing only: if the URL is nil, this placeholder will be returned any time Store.fileURL(for:) is invoked to get the audio file for a recording. With the store constructed, we now need a view controller hierarchy that uses this store:

```
func constructTestingViews(store: Store,
  navDelegate: UINavigationControllerDelegate)
  -> (UIStoryboard, AppDelegate, UISplitViewController,
  UINavigationController, FolderViewController)
{
  let storyboard = UIStoryboard(name: "Main", bundle: Bundle.main)
  let navigationController =
    storyboard.instantiateViewController(withIdentifier: "navController")
    as! UINavigationController
  navigationController.delegate = navDelegate

  let rootFolderViewController = navigationController.viewControllers.first
    as! FolderViewController
  rootFolderViewController.folder = store.rootFolder
  rootFolderViewController.loadViewIfNeeded()
  // ...
  window.makeKeyAndVisible()
  return (storyboard, appDelegate, splitViewController,
    navigationController, rootFolderViewController)
}
```

The above shows the construction of the master navigation controller and the root view controller. The function goes on to similarly construct the detail navigation controller, the play view controller, the split view controller, the window, and the app delegate — a full user interface stack. Constructing all the way up to the window and then calling `window.isHidden = false` is necessary, because otherwise numerous animated and presented actions won't occur.

Notice how the navigation controller's delegate is set to the `navDelegate` parameter (which will be an instance of the `FolderViewControllerTests` class that runs the tests) and the root folder view controller's folder is set to the testing store's `rootFolder`.

We call the above construction functions from the test's `setUp` method to initialize the members on the `FolderViewControllerTests` instance. With that done, we can write our tests.

Integration tests usually require some degree of configuring the initial environment before an action is performed and the result is measured. With application integration tests, measuring the result can range from trivial (accessing a readable property on one of the objects in the configured environment) to very difficult (multi-step asynchronous interactions with the Cocoa framework).

An example of a relatively easy test is testing the commitEditing action for deleting table view rows:

```
func testCommitEditing() {
   // Verify that the action we will invoke is connected
   let dataSource = rootFolderViewController.tableView.dataSource
      as? FolderViewController
   XCTAssertEqual(dataSource, rootFolderViewController)

   // Confirm item exists before
   XCTAssertNotNil(store.item(atUUIDPath: [store.rootFolder.uuid, uuid3]))
   // Perform the action
   rootFolderViewController.tableView(rootFolderViewController.tableView,
      commit: .delete, forRowAt: IndexPath(row: 1, section: 0))
   // Assert item is gone afterward
   XCTAssertNil(store.item(atUUIDPath: [store.rootFolder.uuid, uuid3]))
}
```

The above test verifies that the root view controller is correctly configured as the data source, invokes the data source method tableView(_:commit:forRowAt:) directly on the root view controller, and confirms that this deletes the item from the model.

When animations or potential simulator-dependent changes are involved, the tests get more complicated. Testing that selecting a recording row in the folder view controller correctly displays that recording in the split view's detail position is the most complex test in the Recordings app. The test needs to handle the difference between the split view controller's collapsed and uncollapsed states, and it needs to wait for potential navigation controller push actions to complete:

```
func testSelectedRecording() {
   // Select the row so `prepare(for:sender:)` can read the selection
   rootFolderViewController.tableView.selectRow(at: IndexPath(row: 1, section: 0),
      animated: false, scrollPosition: .none)
   // Handle the collapsed or uncollapsed split view controller
   if self.splitViewController.viewControllers.count == 1 {
      ex = expectation(description: "Wait for segue")
```

```
        // Trigger the transition
        rootFolderViewController.performSegue(withIdentifier: "showPlayer",
          sender: nil)
        // Wait for the navigation controller to push the collapsed detail view
        waitForExpectations(timeout: 5.0)
        // Traverse to the `PlayViewController`
        let collapsedNC = navigationController.viewControllers.last
          as? UINavigationController
        let playVC = collapsedNC?.viewControllers.last as? PlayViewController
        // Test the result
        XCTAssertEqual(playVC?.recording?.uuid, uuid3)
    } else {
        // Handle the uncollapsed state...
    }
}
```

The expectation is fulfilled when the master navigation controller reports that it has pushed a new view controller. This change to the master navigation controller takes place when the detail view is collapsed onto the master view (as happens in a compact display — i.e. every iPhone display except landscape on the iPhone Plus):

```
func navigationController(_ navigationController: UINavigationController,
  didShow viewController: UIViewController, animated: Bool) {
  ex?.fulfill()
  ex = nil
}
```

While these integration tests get the job done — the tests we've written for the folder view controller cover roughly 80 percent of its lines and test all important behaviors — this testSelectedRecording test reveals that correctly writing integration tests can require significant knowledge about how the Cocoa framework operates.

Discussion

MVC has the advantage that it is the architectural pattern for iOS development with the lowest friction. Every class in Cocoa is tested under MVC conditions. Features like storyboards, which rely on heavy integration between the frameworks and classes, are more likely to work smoothly with a program when using MVC. When looking on the internet, you'll find more examples following the MVC pattern than examples of any other design pattern. Additionally, MVC often has the least code and the least design overhead of all patterns.

As the most used of all the patterns we explore in this book, MVC also has the most clearly understood shortcomings — in particular, MVC is associated with two common problems.

Observer Pattern Failures

The first of these problems is when the model and view fall out of synchronization. This occurs when the observer pattern around the model is imperfectly followed. A common mistake is to read a model value on view construction and not subscribe to subsequent notifications. Another common mistake is to change the view hierarchy at the same time as changing the model, thereby assuming the result of the change instead of waiting for the observation, even when the model might reject the change afterward. These kinds of mistakes can cause the views to go out of sync with the model, and unexpected behavior follows.

Unfortunately, Cocoa doesn't provide any checks or built-in mechanisms to verify a correct implementation of the observer pattern. The solution is to be strict in applying the observer pattern: when reading a model value, it's also necessary to subscribe to it. Other architectures (such as TEA or MAVB) combine the initial reading and subscribing into a single statement, making it impossible to make an observation mistake.

Massive View Controllers

The second problem associated with MVC — *Cocoa* MVC in particular — is that it often leads to large view controllers. View controllers have view layer responsibilities (configuring view properties and presenting views), but they also have controller layer responsibilities (observing the model and updating views), and they can end up with model layer responsibilities (fetching, transforming, or processing data). Combined with their central role in the architecture, this makes it easy to carelessly assign every responsibility to the view controller, rapidly making a program unmanageable.

There's no clear limit for how big a view controller can reasonably be. Xcode starts showing obvious pauses when opening, browsing, and viewing Swift files of more than

2,000 lines, so it's probably better to avoid anything near this length. Most screens display 50 lines of code or less, so a dozen screens worth of scrolling start to make it visually difficult to find code.

But the strongest argument against large view controllers isn't the lines of code so much as the amount of state. When the entire file is a single class — like a view controller — any mutable state will be shared across all parts of the file, with each function needing to cooperatively read and maintain the state to prevent inconsistency. Factoring the maintenance of state into separate interfaces forces better consideration of data dependencies and limits the potential amount of code that must cooperatively obey rules to ensure consistency.

Improvements

Observer Pattern

For the base MVC implementation, we opted to broadcast model notifications using Foundation's NotificationCenter. We chose this because we wanted to write the implementation using basic Cocoa, with limited use of libraries or abstractions.

However, the implementation requires cooperation in many locations to correctly update values. The folder view controller gets its folder value when the parent folder view sets it during prepare(for:sender:):

```
guard
  let folderVC = segue.destination as? FolderViewController,
  let selectedFolder = selectedItem as? Folder
else { fatalError() }
folderVC.folder = selectedFolder
```

At a later time, the folder view controller observes notifications on the model to get changes to this value:

```
override func viewDidLoad() {
  super.viewDidLoad()
  // ...
  NotificationCenter.default.addObserver(self,
      selector: #selector(handleChangeNotification(_:)),
      name: Store.changedNotification, object: nil)
}
```

And in the notification handler, if the notification matches the current folder, the folder is updated:

```
@objc func handleChangeNotification(_ notification: Notification) {
   // Handle changes to the current folder
   if let item = notification.object as? Folder, item === folder {
      let reason = notification.userInfo?[Item.changeReasonKey] as? String
      if reason == Item.removed, let nc = navigationController {
         nc.setViewControllers(nc.viewControllers.filter { $0 !== self },
            animated: false)
      } else {
         folder = item
      }
   }
   // ...
```

It would be better to have an observing approach that didn't have the timing gap between the initial setting of the folder and the establishment of the observer in viewDidLoad. It would also be better if there was only one location where the folder needed to be set.

It is possible to use key-value observation (KVO) instead of notifications, but the need to observe multiple different key paths in most cases (for example, observing the parent folder's children as well as the current child) prevents KVO from actually making handling more robust. Since it also requires that every observed property be declared dynamic, it is dramatically less popular in Swift than in Objective-C.

The simplest way to improve the observer pattern is to wrap NotificationCenter and implement the *initial* concept that KVO includes. This concept sends the initial value as soon as the observation is created, allowing us to combine the concepts of *set initial value* and *observe subsequent values* into a single pipeline.

This also allows us to replace both the viewDidLoad and the handleChangeNotification with the following:

```
observations += Store.shared.addObserver(at: folder.uuidPath) {
  [weak self] (folder: Folder?) in
  guard let strongSelf = self else { return }
  if let f = folder { // change
    strongSelf.folder = f
  } else { // deletion
    strongSelf.navigationController.map {
      $0.setViewControllers($0.viewControllers.filter { $0 !== self },
        animated: false)
    }
  }
}
```

It's not dramatically less code, but the self.folder value is set in just one location (inside the observation callback), so we can reduce the number of change paths in our code. Furthermore, there's no more need for dynamic casting in user code, and there's also no gap between the initial setting of the value and the establishing of the observation: the initial value is not the real data but an identifier — the only way we need to access the real data is through the observation callback.

This type of addObserver implementation has the advantage that it can be implemented as an extension to the Store type without any changes to the store itself:

```
extension Store {
  func addObserver<T: AnyObject>(at uuidPath: [UUID],
    callback: @escaping (T?) -> ()) -> [NSObjectProtocol]
  {
    guard let item = item(atUUIDPath: uuidPath) as? T else {
      callback(nil)
      return []
    }
    let o = NotificationCenter.default.addObserver(
      forName: Store.changedNotification,
      object: item, queue: nil) { notification in
      if let item = notification.object as? T, item === item {
        let reason = notification.userInfo?[Item.changeReasonKey] as? String
        if reason == Item.removed {
          return callback(nil)
        } else {
                callback(item)
            }
        }
    }
    callback(item)
    return [o]
  }
}
```

The store still sends the same notifications; we're just subscribing to the notifications a different way, preprocessing the notification data (like testing for the Item.changeReasonKey), and handling other boilerplate code so that the work in each view controller is simpler.

In the chapter on MVC+VS, we show how to take this approach further. MVC+VS is a variant of MVC in which all model data is read using a similar pattern: we receive the initial value and subsequent changes through the same callback.

The Massive View Controller Problem

Very large view controllers often perform work unrelated to their primary role (observing the model, presenting views, providing them with data, and receiving their actions); or they should be broken into multiple controllers that each manage a smaller section of the view hierarchy; or the interfaces and abstractions are failing to encapsulate the complexity of the tasks in a program and the view controllers are cleaning up the mess.

In many cases, the best way to approach this problem is to proactively move as much functionality as possible into the model layer. Sorting, fetching, and processing of data are common functions that end up in the controller because they're not part of the persistent state of the application, but they still relate to an application's data and domain logic and are better placed in the model.

The largest view controller in the Recordings app is the FolderViewController. It is around 130 lines of code, making it far too small for us to see any significant problems. Instead, we'll take a brief look at some large view controllers from popular iOS projects on GitHub. Please note, we are not trying to disparage these projects in any way; we are simply using the open source nature of them to look at why a view controller might grow to thousands of lines of code. Also be aware that the line-count numbers we'll discuss are generated by cloc, which ignores whitespace and comments.

Wikipedia's PlacesViewController

The version of the file reviewed for this book is 2c1725a of PlacesViewController.swift.

This file is 2,326 lines long. The view controller includes the following roles:

1. Configuring and managing the map view, which displays results (400 lines)
2. Getting user location with the location manager (100 lines)
3. Performing searches and gathering results (400 lines)
4. Grouping results for display in the visible area of the map (250 lines)
5. Populating and managing the search suggestion table view (500 lines)
6. Handling the layout of overlays like the suggestion table view (300 lines)

This example demonstrates the classic trifecta of massive view controller causes:

1. More than one major view being managed (map view and suggestion table view).
2. Creating and executing asynchronous tasks (like fetching the user location), although the view controller is only interested in the result of the task (in this case, the user location).
3. Model/domain logic (searching and processing results) performed in the controller layer.

We can simplify the view requirements in this scene by separating the major views into their own, smaller controllers. They don't even need to be instances of `UIViewController` — they could just be child objects owned by the scene. The only work left in the parent view controller might then be integration and layout (and complex layouts can easily be factored out of the view controller as well).

We can create utility classes to perform asynchronous tasks — like getting user location information — and the only code required in the controller would be the construction of the task and the callback closure.

From an application design perspective, the biggest problem here is the model/domain logic in the controller layer. This code lacks an actual model to perform searches and gather search results. Yes, there is a `dataStore` object used to hold this information, but it has no abstraction around it, so it is not helpful on its own. The view controller does all the actual search and data processing itself — these tasks should be handled in another location. Even work like the grouping of results for the visible area could be performed by the model or a transformation object between the model and the view controller.

WordPress' AztecPostViewController

The version of the file reviewed for this book is 6dcf436 of AztecPostViewController.swift.

This file is 2,703 lines long. The view controller includes the following roles:

1. Constructing subviews in code (300 lines)
2. Auto Layout constraints and title placement (200 lines)
3. Managing the article publishing process (100 lines)
4. Setting up child view controllers and handling their results (600 lines)
5. Coordinating, observing, and managing input to the text view (300 lines)
6. Displaying alerts and the alert contents (200 lines)
7. Tracking media uploads (600 lines)

Media uploads is a domain service that could easily be moved into its own model layer service.

Meanwhile, 75 percent of the remaining code could be eliminated by improving the interfaces to other components in the program.

None of the models, services, or child view controllers that this AztecPostViewController deals with can be used in a single line. Displaying an alert takes half a dozen lines in multiple locations. Running a child view controller takes 20 lines of setup and 20 lines of handling after completion. Even though the text view is the custom `Aztec.TextView`, there are hundreds of lines in the view controller tweaking its behavior.

These are all examples where the view controller is being used to patch the behavior of other components that are failing to complete their own jobs. Instead, these behaviors should all be baked into the components where possible. When we can't change the behavior of a component, we can write a wrapper around the component rather than putting this logic in the view controller.

Firefox's BrowserViewController

The version of the file reviewed for this book is 98ec57c of BrowserViewController.swift.

This file is 2,209 lines long. The view controller includes more than 1,000 lines of delegate implementations, including: `URLBarDelegate`, `TabToolbarDelegate`, `TabDelegate`, `HomePanelViewControllerDelegate`, `SearchViewControllerDelegate`, `TabManagerDelegate`, `ReaderModeDelegate`, `ReaderModeStyleViewControllerDelegate`, `IntroViewControllerDelegate`, `ContextMenuHelperDelegate`, and `KeyboardHelperDelegate`.

What are these and what do they do?

Basically, the `BrowserViewController` is the top-level view controller in the program, and these delegate implementations represent lower-level view controllers using the `BrowserViewController` to relay actions around the program.

Yes, it is the controller layer's responsibility to relay actions around the program — so this isn't a case of model/domain responsibilities spilling out into the wrong layer — but many of these delegate actions are unrelated to the view managed by the `BrowserViewController` (they merely want to access other components or state stored on the `BrowserViewController`).

Instead of putting this responsibility onto a view controller that already has plenty of other responsibilities, these delegate callbacks could all be relocated onto a coordinator or other abstract (non-view) controller dedicated to handling relays within the controller layer.

Code Instead of Storyboards

Rather than using storyboards, we can choose to define our view hierarchy in code. This change gives us more control over the construction phase, and one of the biggest advantages is that we will have better control over dependencies.

For example, when using storyboards, there is no way to guarantee that all necessary properties for a view controller are set in prepare(for:sender:). Recall that we've used two different techniques for passing model objects: a default value (as in the folder view controller), and an optional type (as in the play view controller). Neither of these techniques guarantees that the object is passed; if we forget to do this, they silently continue with either the wrong value or an empty value.

When we move away from storyboards, we gain more control over the construction process, and we can let the compiler ensure that the necessary parameters are passed in. Note that it is not necessary to completely dispose of storyboards. We could start by removing all segues and performing them manually instead. To construct a folder view controller, we can add a static method:

```
extension FolderViewController {
 static func instantiate(_ folder: Folder) -> FolderViewController {
   let sb = UIStoryboard(name: "Main", bundle: nil)
   let vc = sb.instantiateViewController(withIdentifier: "folderController")
     as! FolderViewController
   vc.folder = folder
   return vc
 }
}
```

When we create a folder view controller using the instantiate method, the compiler helps us and tells us we need to provide the folder; it is not possible to forget to provide it. Ideally, instantiate would be an initializer, but that's only possible if we move away from storyboards completely.

We can apply the same technique to view construction as well by adding convenience initializers to view classes or writing functions to construct specific view hierarchies. In general, eschewing storyboards allows us to use all language features in Swift: generics (e.g. to configure a generic view controller), first-class functions (e.g. to style a view or set callbacks), enums with associated values (e.g. to model exclusive states), and so on. At the time of writing, storyboards don't support these features.

Reusing Code with Extensions

To share code between view controllers, a common solution is to create a superclass containing the shared functionality. A view controller then gains that functionality by subclassing. This technique can work well, but it has a potential downside: we can only pick a single superclass for our new class — for example, it's not possible to inherit from both UIPageViewController and UITableViewController. This technique also often leads to what we call the *god view controller*: a shared superclass that contains all shared functionality in the project. Such a class often becomes so complex that it decreases maintainability.

Another way to share code between view controllers is by using extensions. Methods that occur in multiple view controllers can sometimes be added as an extension on UIViewController instead. That way, all view controllers gain that method. For example, we added a convenience method to UIViewController that displays modal text alerts.

For an extension to be useful, it's often necessary that the view controller has some specific capabilities. For example, the extension might require that a view controller has an activity indicator present, or that a view controller has certain methods available. We can capture these capabilities in a protocol. As an example, we can share keyboard handling code, resizing a view when the keyboard shows or hides. If we use Auto Layout, we can specify that we expect the capability of having a resizable bottom constraint:

```swift
protocol ResizableContentView {
  var resizableConstraint: NSLayoutConstraint { get }
}
```

Then we can add an extension to every UIViewController that implements this protocol:

```swift
extension ResizableContentView where Self: UIViewController {
  func addKeyboardObservers() {
    // ...
  }
}
```

Now, any view controller that conforms to ResizableContentView also gains the addKeyboardObservers method. We can use the same technique in other cases where we want to share code without subclassing.

Reusing Code with Child View Controllers

Child view controllers are another option for sharing code between view controllers. For example, if we want to show a small player at the bottom of the folder view controller, we can add a child view controller to the folder view controller so that the player logic is contained and doesn't clutter up the folder view controller. This is easier and more maintainable than duplicating the code within the folder view controller.

If we have a single view controller with two distinct states, we could also separate it into two view controllers (one for each state) and use a container view controller to switch between the two child view controllers. For example, we could split up the play view controller into two separate view controllers: one that shows the "No recording selected" text, and another that displays a recording. A container view controller can then switch between the two depending on the state. There are two advantages to this approach: first, the empty view controller can be reused if we make the title (and other properties) configurable. Second, the play view controller won't have to deal with the case where the recording is nil; we only create and display it when we have a recording.

Extracting Objects

Most large view controllers have many roles and responsibilities. Often, a role or responsibility can be extracted into a separate object, although it's not always easy to see this refactoring. We find it helpful to distinguish between coordinating controllers and mediating controllers (as defined by Apple). A coordinating controller is application specific and generally not reusable (for example, almost all view controllers are coordinating controllers).

A mediating controller is a reusable controller object, which is configured for a specific task. For example, the AppKit framework provides classes like NSArrayController or NSTreeController. On iOS, we can build similar components. Often, a protocol conformance (such as the folder view controller conforming to UITableViewDataSource) is a good candidate for a mediating controller. Pulling out these "conformances" into separate objects can be an effective way to make a view controller smaller. As a first step, we can extract the table view's data source in the folder view controller without making modifications to the implementation of the methods:

```
class FolderViewDataSource: NSObject, UITableViewDataSource {
   var folder: Folder

   init(_ folder: Folder) {
      self.folder = folder
   }

   func numberOfSections(in tableView: UITableView) -> Int {
      return 1
   }

   func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
      -> Int
   {
      return folder.contents.count
   }

   func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
      -> UITableViewCell
   {
      let item = folder.contents[indexPath.row]
      let identifier = item is Recording ? "RecordingCell" : "FolderCell"
      let cell = tableView.dequeueReusableCell(withIdentifier: identifier,
         for: indexPath)
      cell.textLabel!.text = "\((item is Recording) ? "🔊" : "📁")  \(item.name)"
      return cell
   }

   func tableView(_ tableView: UITableView,
      commit editingStyle: UITableViewCellEditingStyle,
      forRowAt indexPath: IndexPath)
   {
      folder.remove(folder.contents[indexPath.row])
   }

   func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath)
```

```
        -> Bool
    {
        return true
    }
}
```

Then, in the folder view controller itself, we set the table view's data source to our new data source:

```
lazy var dataSource = FolderViewDataSource(folder)

override func viewDidLoad() {
    super.viewDidLoad()
    tableView.dataSource = dataSource
    // ...
}
```

We also need to take care to observe the view controller's folder and change the data source's folder in response. At the price of some extra communication, we have separated the view controller into two components. In this case, the separation doesn't have too much overhead, but when the two components are tightly coupled and need a lot of communication or share a lot of state, the overhead might be so big that it only makes things more complicated.

If we have multiple similar objects, we might be able to generalize them. For example, in the `FolderViewDataSource` above, we could change the stored property from `Folder` to `[Item]` (an `Item` is either a folder or a recording). And as a logical next step, we can make the data source generic over the `Element` type and move the `Item`-specific logic out. This means that cell configuration (through the `configure` parameter) and deletion logic (through the `remove` parameter) are now passed in from the outside:

```
class ArrayDataSource<Element>: NSObject, UITableViewDataSource {
    // ...
    init(_ contents: [Element],
           identifier: @escaping (Element) -> String,
           remove: @escaping (_ at: Int) -> (),
           configure: @escaping (Element, UITableViewCell) -> ()) {
        // ...
    }
    // ...
}
```

To configure it for our folders and recordings, we need the following code:

```
ArrayDataSource(folder.contents,
    identifier: { $0 is Recording ? "RecordingCell" : "FolderCell" },
    remove: { [weak self] index in
        guard let folder = self?.folder else { return }
        folder.remove(folder.contents[index])
    }, configure: { item, cell in
        cell.textLabel!.text = "\((item is Recording) ? "🔊" : "📁")  \(item.name)"
    })
```

In our small example app, we don't gain much by making our code so generic. However, in larger apps, this technique can help reduce duplicated code, allow type-safe reuse, and make view controllers simpler.

Simplifying View Configuration Code

If the view controller is constructing and updating a lot of views, it can be helpful to pull this view configuration code out. Especially in the case of "set and forget," where there isn't two-way communication, doing this can simplify our view controller. For example, when we have a complicated tableView(_:cellForRowAtIndexPath:), we can move some of that code out of the view controller:

```
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
    let item = folder.contents[indexPath.row]
    let cell = tableView.dequeueReusableCell(withIdentifier: identifier,
        for: indexPath)
    cell.configure(for: item) // Extracted
    return cell
}
```

We can then move all that logic onto UITableViewCell or a subclass:

```
extension UITableViewCell {
  func configure(for item: Item) {
    textLabel!.text = "\((item is Recording) ? "🔊" : "📁")  \(item.name)"
  }
}
```

In our case, extracting cell configuration made the view controller marginally simpler, because the original code was just one line. However, if there is cell configuration code that spans multiple lines, it might be worth factoring that code out. We can also use this pattern to share configuration and layout code between different view controllers. As an added benefit, it's easy to see that `configure(for:)` doesn't depend on any of the state in the view controller: all state is passed in through parameters, and as such, it's easy to test.

Conclusion

MVC is the simplest and most commonly used of the patterns we will discuss in this book. The other application design patterns in this book all represent — to varying degrees — a break from convention. Unless you know you want to choose a less conventional path for your project, you should probably start your projects in MVC.

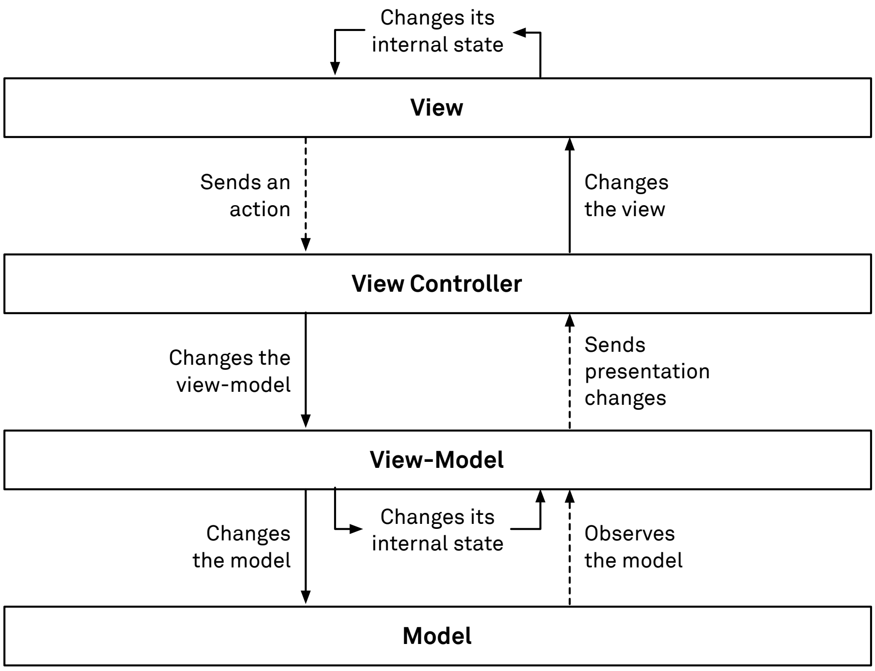
MVC is sometimes discussed in disparaging terms by programmers advocating different architectural patterns. Claims against MVC include: view controllers aren't testable, view controllers grow too large and cumbersome, and data dependencies are difficult to manage. However, after reading this chapter, we hope you've seen that while MVC has its own challenges, it is possible to write clear, concise view controllers that include full tests and cleanly manage their data dependencies.

Still, there are other ways to approach application design that alter where the challenges lie. In the following chapters, we'll show a number of alternative design patterns, each of which solves the problems of application design in a different way. All of those patterns will include ideas you can apply to MVC programs to solve specific problems pragmatically or produce your own hybrid patterns or programming style.

Model-View-ViewModel+ Coordinator

4

Model-View-ViewModel (MVVM) is a pattern that aims to improve upon MVC by moving all model-related tasks (updating the model, observing it for changes, transforming model data for display, etc.) out of the controller layer and into a new layer of objects called view-models. View-models — in common iOS implementations — sit between the model and the view controller:



As with all good patterns, this isn't just about moving code into a new location. The purpose of the new view-model layer is twofold:

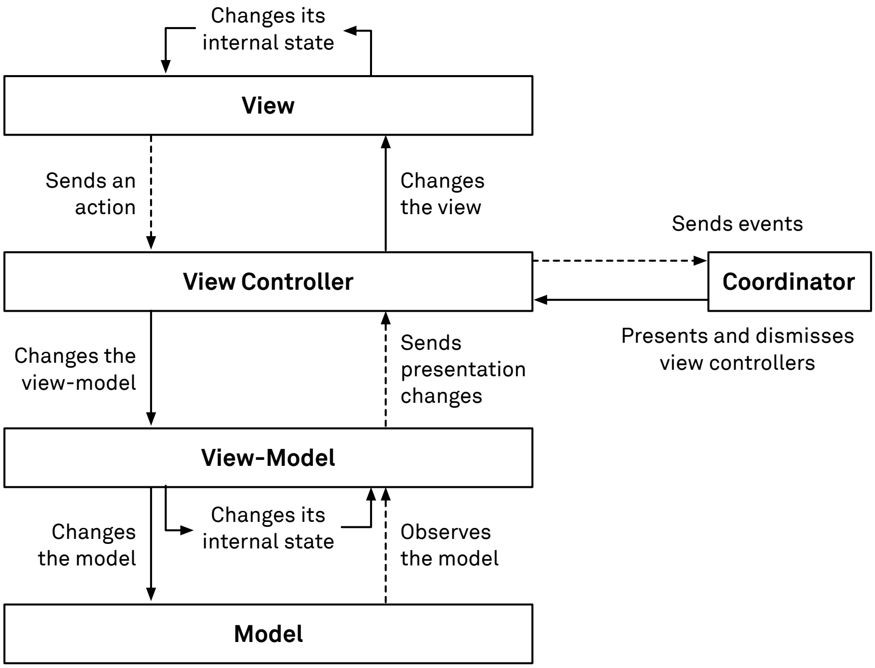
1. To encourage structuring the relationship between the model and the view as a pipeline of transformations.
2. To provide an interface that is independent of the application framework but substantially represents the views' presentation state.

Together, these points address two of the biggest criticisms of MVC. The first reduces the responsibilities of view controllers by moving model-related observation and transformation tasks out of the controller layer. The second provides a clean interface to a scene's view state that can be tested independent of the application framework — as opposed to MVC's integrated testing.

To keep the view synchronized with the view-model, MVVM mandates the use of some form of bindings, i.e. any mechanism designed to keep properties on one object in sync with properties on another object. The view controller constructs these bindings between properties exposed by the view-model and properties on views in the scene that the view-model represents.

> We'll be using reactive programming to implement the bindings, as this is the most common approach for MVVM on iOS. However, reactive programming is not a default inclusion in Cocoa projects, and reactive code can be difficult to read if you're unfamiliar with it. Please see the section below, "A Quick Introduction to Reactive Programming," for help with the terms and concepts.
>
> We've chosen to implement the MVVM variant of our example app using reactive programming because we believe it's putting MVVM's best foot forward. Furthermore, we consider reactive programming beneficial to learn about irrespective of whether or not you end up using it in your code base. That being said, we've included a section later on in this chapter that shows how you can use MVVM with less reliance on reactive programming, or even just with bindings based on Foundation's key-value observing (KVO) mechanism.

Additionally, the MVVM version of our example app introduces a coordinator component. The coordinator is not a mandatory component of MVVM, but it helps relieve the view controllers of two additional responsibilities: managing the presentation of other view controllers, and mediating communication of model data between view controllers. The coordinator is in charge of managing the view controller hierarchy so that the view controllers and view-models only need to be concerned about their particular scenes. The diagram of Model-View-ViewModel+Coordinator (MVVM-C) looks like this:



In our example app, the coordinator is the delegate of view controllers, and view controllers forward navigation actions directly to the coordinator. Some people prefer to forward navigation actions to the view-model first, and then have the coordinator observe the view-model for navigation events. The latter approach is useful if navigation events are dependent on the current view state or model data. Moving the navigation events from the view controller to the view-model also allows for easier testing of this interplay. However, our example app wouldn't have gained any practical advantage from looping navigation events through the view-model, so we opted for the simpler solution.

Exploring the Implementation

The MVVM pattern can be seen as an elaborate version of MVC. Our implementation adds two kinds of components to remove some of the view controller responsibilities: coordinators take on navigation responsibilities, and view-models take on most of the logic that was previously in the view controller. Despite this similarity between the two patterns, there are parts of an MVVM implementation that may look radically different.

The primary reason for the difference in coding style is that communication between the model and the view in MVVM is not a single observer pattern interface, as in MVC, but rather is restructured as a data pipeline, a task which we implement by using reactive programming. Reactive programming builds a data pipeline using a series of transformation stages, which can look very different to building logic with Swift's basic control-flow statements (loops, conditions, method calls).

To limit confusion for those unfamiliar with reactive programming, we're going to summarize the important concepts in reactive programming before looking at how construction, model changes, and view state changes are handled in MVVM.

A Quick Introduction to Reactive Programming

Reactive programming is a pattern for describing data flows between sources and consumers of data as a pipeline of transformations. Data flowing through the pipeline is treated as a sequence and is transformed using functions with names similar to some of Swift's sequence and collection functions, such as `map`, `filter`, and `flatMap`.

Let's look at a quick example. Imagine we have a model object with an optional string property. We want to observe the property and apply the observed values to a label. Assuming the property is declared `dynamic`, we can observe it in a view controller's `viewDidLoad` method using KVO:

```swift
var modelObject: ModelObject!
var obs: NSKeyValueObservation? = nil

override func viewDidLoad() {
  super.viewDidLoad()

  obs = modelObject.observe(\ModelObject.value) { [unowned self] obj, change in
    if case let value?? = change.newValue {
      self.textLabel.string = "Selected value is: \(value)"
    } else {
      self.textLabel.string = "No value selected"
    }
  }
}
```

Other than the need to unwrap two layers of optionals around change.newValue, it’s fairly standard Cocoa programming; we observe the value, and when it changes, we update the text label.

Using RxSwift for reactive programming, we could instead write this:

```
var modelObject: ModelObject!
var disposeBag = DisposeBag()

override func viewDidLoad() {
   super.viewDidLoad()

   modelObject.valueObservable.map { possibleValue -> String in
      if let value = possibleValue {
         return "Selected value is: \(value)"
      } else {
         return "No value selected"
      }
   }.bind(to: self.textLabel.rx.text).disposed(by: disposeBag)
}
```

The differences might initially seem aesthetic:

- → Instead of observing the value property directly, we read a property named valueObservable.
- → Instead of performing all the work in a single callback, we perform the purely data-related transformation in the middle — on its own — using a map transformation.
- → Instead of setting the textLabel.text directly, we set it at the end of the chain, inside the bind(to:) call.

Why is this important?

Instead of having multiple places where the textLabel.text value is set, the text label is referenced just once, at the end. Reactive programming asks us to start at the destination — the *subscriber* of the data — and walk our way backward through the data transformations to the original data dependencies — the *observables*. In doing this, it keeps the three parts of a data pipeline — the observables, the transformations, and the subscriber — separate.

The transformations are both the biggest benefit of reactive programming and the point where the learning curve gets the steepest. You have probably used map with Swift sequences or optionals, and the RxSwift version works in a similar way. Other functions in RxSwift that have Sequence equivalents include filter (same in both), concat (similar to append(contentsOf:)), skip and skipWhile (similar to dropFirst and dropWhile), and take and takeWhile (similar to prefix and prefixWhile).

It is worth discussing the flatMapLatest transformation. This transformation observes its source, and every time the source emits a value, it uses that value to construct, start, or select a new observable. The values emitted by the new observable are emitted from the flatMapLatest result. It might seem confusing, but it lets us subscribe to a second observable based on the state emitted through a first observable.

It's also worth mentioning some of the types in RxSwift:

→ Observable is a stream of values we can transform, subscribe to, or bind to a UI element.

→ PublishSubject is an Observable, but we can also *send* values to it, which will be emitted to observers.

→ ReplaySubject is like PublishSubject, except we can start sending before any observers have connected, and new observers will receive any previously sent values cached in the "replay" buffer.

→ Variable is a wrapper around a settable value and offers an Observable property so we can observe the value each time it is set.

→ Disposable and DisposeBag are used to control the lifetime of one or more subscriptions, respectively. When the disposable is deallocated or explicitly disposed, the subscription ends, and all the observables that were part of the subscription are released.

That's plenty of information to absorb. Perhaps we should move on and look at MVVM.

Construction

MVVM's approach to construction follows a pattern similar to MVC: the controller layer has full knowledge of the structure of the program and uses this knowledge to construct and connect all components. Relative to MVC, there are three key differences:

1. The view-model must be constructed.
2. Bindings between the view-model and views must be established.
3. The view-model (not the controller) owns the model.

To address the first difference, we chose to construct a default view-model within each view controller:

```
class FolderViewController: UITableViewController {
  let viewModel = FolderViewModel()
  // ...
}
```

This particular arrangement is a simple, low-friction approach when using storyboards and segues, since the view-model does not need to be set on the view controller during segues or otherwise set after construction. The view-model is not configured with a reference to a model object at construction; the model object has to be set on the view-model at a later point.

In an alternative arrangement, the view-model on the view controller is declared as an initially `nil` optional, and a fully configured view-model is set on the view controller at a later point. We avoided this approach because not only did it fail to solve any data propagation issues in the Recordings app, but it also added an optional to an otherwise non-optional type. However, if we were to ignore storyboards entirely and use manual view controller initialization, we could pass the view-model in as a parameter to the view controller and achieve the best of both arrangements.

Connecting Views to Initial Data

In the MVVM-C pattern we're using, it is the coordinator's responsibility to set the model objects on the view-models. The coordinator itself is a high-level controller object, which is constructed in the application delegate as the final step in the startup process, so it must ensure that any initially constructed view controllers are provided with data during its construction:

```
@UIApplicationMain
class AppDelegate:
  UIResponder, UIApplicationDelegate, UISplitViewControllerDelegate {

  var coordinator: Coordinator? = nil

  func application(_ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
      [UIApplicationLaunchOptionsKey: Any]?)
    -> Bool
  {
    let splitViewController = window!.rootViewController as! UISplitViewController
    splitViewController.delegate = self
    splitViewController.preferredDisplayMode = .allVisible
    coordinator = Coordinator(splitViewController)
    return true
  }
  // ...
}
```

In its own constructor, the coordinator ensures that the top-level view-models are provided with their initial data:

```
final class Coordinator {
  init(_ splitView: UISplitViewController) {
    // ...
    let folderVC = folderNavigationController.viewControllers.first
      as! FolderViewController
    folderVC.delegate = self
    folderVC.viewModel.folder.value = Store.shared.rootFolder
    // ...
  }
  // ...
}
```

The line folderVC.viewModel.folder.value = Store.shared.rootFolder is the most important, as this is where the root FolderViewModel receives its data. It's also the first line of code using the RxSwift framework; the folder property on the folder view-model is an RxSwift Variable:

```
class FolderViewModel {
  let folder: Variable<Folder>
  // ...
}
```

Changing a Variable's value has two effects: the first is that the folder view-model's folder.value property will return the value we have set. The second is that any observers of this variable will also receive the new value.

To explore how the views ultimately receive this data, we need to look at the implementation of the folder view-model. To observe its folder variable, the view-model constructs a dependent folderUntilDeleted observable on initialization:

```
// FolderViewModel
private let folderUntilDeleted: Observable<Folder?>

init(initialFolder: Folder = Store.shared.rootFolder) {
  folder = Variable(initialFolder)
  folderUntilDeleted = folder.asObservable()
    // Every time the folder changes
    .flatMapLatest { currentFolder in
      // Start by emitting the initial value
      Observable.just(currentFolder)
        // Reemit the folder every time a non-delete change occurs
        .concat(currentFolder.changeObservable.map { _ in currentFolder })
        // Stop when a delete occurs
        .takeUntil(currentFolder.deletedObservable)
        // After a delete, set the current folder back to `nil`
        .concat(Observable.just(nil))
    }.share(replay: 1)
}
```

We added a comment to each line, so if you're not familiar with reactive programming, please read the comments for a step-by-step summary of this code. Reactive programming builds logic from sequences of its own transformations. These transformations encode significant logic in a highly efficient syntax, but needing to learn the common reactive programming transformations adds a steep learning curve to reading reactive programming code.

The gist of the code above is: we combine the folder observable with other model-derived data observations that might affect the logic of our view. The result is a folderUntilDeleted observable that will update correctly when the underlying folder object is mutated and will set itself to nil if the underlying folder object is deleted from the store.

The folderUntilDeleted observable is not observed directly by the view controller. The purpose of the view-model is to transform all data into the correct format so it is ready to be directly bound to the view. In this case, it means extracting the required text or other properties from the folder object emitted by the folderUntilDeleted observable.

For example, the title displayed in the navigation bar at the top of the FolderViewController is prepared in the FolderViewModel. Anytime the folderUntilDeleted observable emits a new value, this observable emits a new value too:

```
// FolderViewModel
var navigationTitle: Observable<String> {
   return folderUntilDeleted.map { folder in
      guard let f = folder else { return "" }
      return f.parent == nil ? .recordings : f.name
   }
}
```

The observable output of this navigationTitle property is bound to the view controller's title by the following line in the FolderViewController's viewDidLoad method:

```
class FolderViewController: UITableViewController {
   override func viewDidLoad() {
      super.viewDidLoad()
      viewModel.navigationTitle.bind(to: rx.title).disposed(by: disposeBag)
      // ...
   }
}
```

Thus we have the full pipeline of data:

1. The coordinator sets the initial model objects on the view-model of each view controller.
2. The view-model combines the set value with other model data and observations.
3. The view-model transforms the data into the precise format required for the view.
4. The view controller uses bind(to:) to connect these prepared values to each view.

Applying State Restoration Data

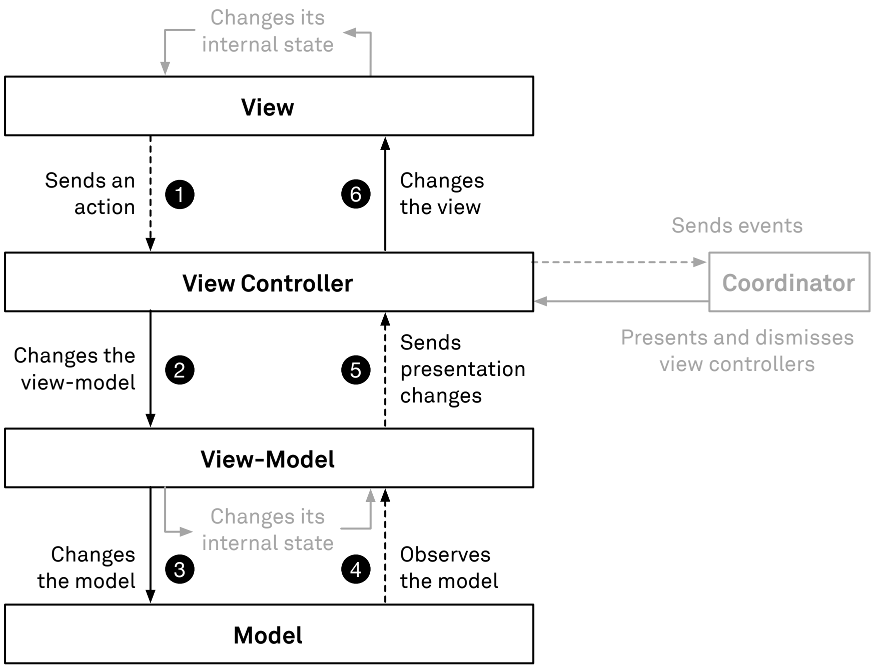
MVVM largely follows the same state restoration approach as MVC, except that the data stored for each view controller comes from the view-model instead of the view controller itself. Therefore, the implementations of encodeRestorableState and decodeRestorableState on the folder view controller are almost identical to the MVC versions:

```
// FolderViewController
override func encodeRestorableState(with coder: NSCoder) {
   super.encodeRestorableState(with: coder)
   coder.encode(viewModel.folder.value.uuidPath, forKey: .uuidPathKey)
}

override func decodeRestorableState(with coder: NSCoder) {
   super.decodeRestorableState(with: coder)
   if let uuidPath = coder.decodeObject(forKey: .uuidPathKey) as? [UUID],
      let folder = Store.shared.item(atUUIDPath: uuidPath) as? Folder
   {
      self.viewModel.folder.value = folder
   } else {
      if var controllers = navigationController?.viewControllers,
         let index = controllers.index(where: { $0 === self })
      {
         controllers.remove(at: index)
         navigationController?.viewControllers = controllers
      }
   }
}
```

The only difference is that the code uses viewModel.folder.value instead of self.folder. Otherwise, it's identical to the MVC code.

Changing the Model

The event loop from view to model and back in MVVM-C follows a similar path through the layers when compared to MVC, with the addition of the mediating view-model inserted between the view controller and the model:



Below we'll look at the implementation of the steps involved to propagate the deletion of a folder from the table view to the model and back up to the view.

Step 1: Table View Sends Action

In our MVVM implementation, the table view's data source is not configured in the storyboard, because we use RxSwift's table view extensions to provide the table view with its data. Therefore, what normally would be a call to the data source's tableView(_:commit:forRowAt:) method is encapsulated as an event to the table view's modelDeleted observable.

Step 2: View Controller Changes View-Model

In MVVM, the view controller doesn't call into the model layer itself. Instead, it outsources this work to its view-model. As mentioned in the first step, we get notified about table view deletions via Rx's modelDeleted observable on the table view. We subscribe to this observable and call the view-model's deleteItem method:

```
// FolderViewController
override func viewDidLoad() {
  // ...
  tableView.rx.modelDeleted(Item.self)
    .subscribe(onNext: { [unowned self] in
      self.viewModel.deleteItem($0)
    }).disposed(by: disposeBag)
}
```

Step 3: View-Model Changes Model

Inside the view-model, the change is directly propagated to the model layer:

```
// FolderViewModel
func deleteItem(_ item: Item) {
  folder.value.remove(item)
}
```

The remove method on the Folder class removes the specified item from its children and calls save on the store to persist the changes (this is identical to the MVC version):

```
// Folder
func remove(_ item: Item) {
  guard let index = contents.index(where: { $0 === item }) else { return }
  item.deleted()
  contents.remove(at: index)
  store?.save(item, userInfo: [
        Item.changeReasonKey: Item.removed,
        Item.oldValueKey: index, Item.parentFolderKey: self
    ])
}
```

Saving the changes triggers a model notification, which we'll use in the next step.

Step 4: View-Model Observes Model Changes:

As we outlined in the construction section above, the view-model observes its folder variable for changes, and this is combined with the model notifications into folderUntilDeleted. Whenever the folder's contents change, the view-model needs to update its representation of these contents that the table view binds to. This is done in the folderContents observable:

```
// FolderViewModel
var folderContents: Observable<[AnimatableSectionModel<Int, Item>]> {
    return folderUntilDeleted.map { folder in
        guard let f = folder else {
            return [AnimatableSectionModel(model: 0, items: [])]
        }
        return [AnimatableSectionModel(model: 0, items: f.contents)]
    }
}
```

folderContents emits the current contents of the folder in a way that can be bound to the table view, as we'll see in the next step.

Steps 5 & 6: View Controller Observes View-Model and Updates the Views

In viewDidLoad, the view controller binds the view-model's folderContents property to the table view's data source using RxSwift's table view extensions (part of the RxDataSources library):

```
// FolderViewController
override func viewDidLoad() {
    // ...
    viewModel.folderContents.bind(
        to: tableView.rx.items(dataSource: dataSource)
    ).disposed(by: disposeBag)
}
```

This is all we have to do in order to keep the table view in sync with the underlying data. Under the hood, Rx's table view extensions take care of all the table view calls like `insertRows` and `deleteRows`.

Changing the View State

In MVVM-C, part of the view state — the transient state of the user interface — is implicit in the view and view controller hierarchy, while other parts are explicitly represented by view-models.

In theory, a view-model is supposed to be a representation of the full state of the view within a particular scene. However, in common practice, a view-model only represents the view state affected by view actions. For example, while the current scroll position is clearly view state, it's usually not represented by the view-model; it's neither dependent on any view-model properties, nor do other parts of the view-model depend on it. This makes the overall representation of view state in the view-model equivalent to the view state that would typically be held by properties on a view controller subclass in MVC.

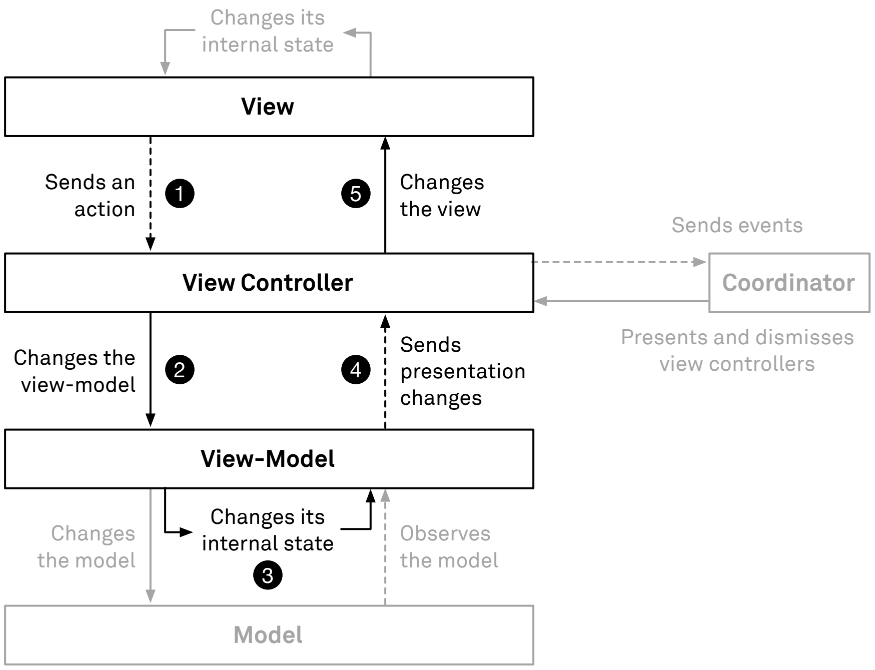
Below, we'll look at the implementation details for two distinct examples of modifying the view state:

1. Changing the title of the play button in response to tapping it.
2. Pushing a folder view controller in response to selecting a subfolder.

In the first example, we only change a property of an existing view (the button's title), whereas in the second example, we need to modify the view and the view controller hierarchy.

Example 1: Updating the Play Button

If a view state change doesn't affect the view controller hierarchy, the coordinator isn't involved. The view controller that receives the user action forwards it to the view-model, and reactive bindings propagate necessary view updates from the view-model to the views. This is the path taken in the first example of updating the play button's title:



Step 1: The Button Sends an Action to the View Controller

The first step works the same as in the MVC variant of the example app: the play button is connected to the play method on the play view controller via Interface Builder, so play is called when the user taps the play button.

Step 2: The Play View Controller Changes the View-Model

The play method makes only a single call by sending a new event into the view-model's togglePlay property (a PublishSubject):

```
// PlayViewController
@IBAction func play() {
    viewModel.togglePlay.onNext(())
}
```

Step 3: The Play View-Model Changes Its Internal State

The call to onNext on the view-model's togglePlay property changes the audio player's state from playing to paused or vice versa. Whenever the state of the audio player changes, a new value is sent into the view-model's internal playState observable. From this observable, we derive the playButtonTitle observable, which maps the possible states to their respective button titles:

```
// PlayViewModel
var playButtonTitle: Observable<String> {
    return playState.map { s in
        switch s?.activity {
        case .playing?: return .pause
        case .paused?: return .resume
        default: return .play
        }
    }
}
```

Steps 4 & 5: The Play View Controller Observes the View-Model and Updates the View

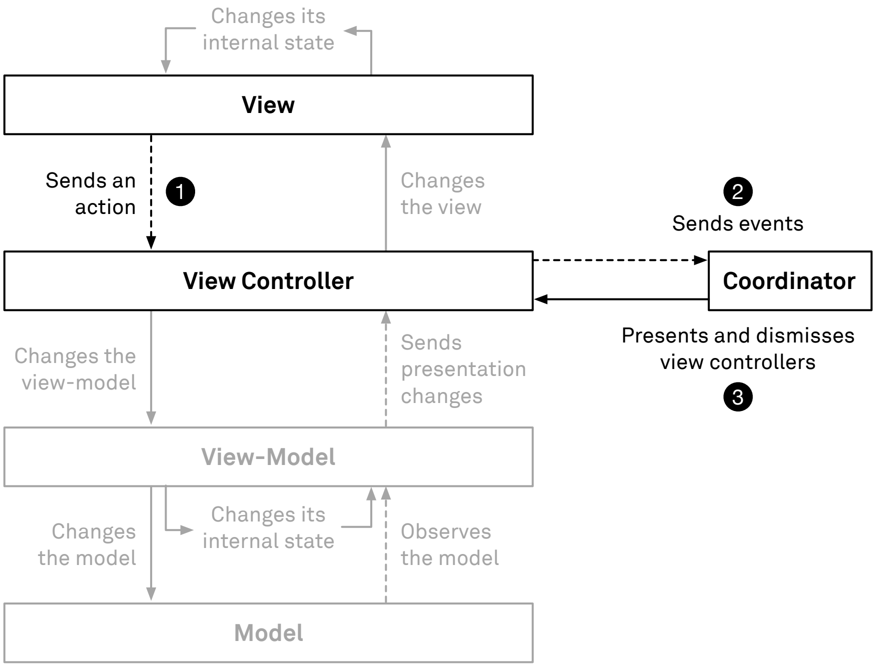
In the play view controller's viewDidLoad, we set up a reactive binding from the view-model's playButtonTitle observable to the play button's title:

```
// PlayViewController
override func viewDidLoad() {
    // ...
    viewModel.playButtonTitle.bind(to: playButton.rx.title(for: .normal))
        .disposed(by: disposeBag)
}
```

This binding observes the view-model's play button title and propagates it to the actual button view.

Example 2: Pushing a Folder View Controller

When we change the view controller hierarchy — as in this example — the coordinator comes into play. The view controller receiving the user action delegates the work of updating the view controller hierarchy to the coordinator:



Steps 1 & 2: The Folder View Controller Receives the Selection Action and Calls the Coordinator

In the folder view controller's viewDidLoad, we subscribe to the table view's selection using RxSwift's table view extensions. When we receive a selection event, we forward it to the folder view controller's delegate (which is the coordinator):

```
// FolderViewController
override func viewDidLoad() {
  // ...
```

```
    tableView.rx.modelSelected(Item.self)
        .subscribe(onNext: { [unowned self] in
            self.delegate?.didSelect($0)
        }).disposed(by: disposeBag)
}
```

Step 3: The Coordinator Pushes the New Folder View Controller

The coordinator's didSelect method checks whether or not the selected item is a folder. If so, it instantiates a new folder view controller from the storyboard and pushes it onto the navigation stack:

```
extension Coordinator: FolderViewControllerDelegate {
    func didSelect(_ item: Item) {
        switch item {
        case let folder as Folder:
            let folderVC = storyboard.instantiateFolderViewController(
                with: folder, delegate: self)
            folderNavigationController.pushViewController(folderVC,
                animated: true)
        // ...
        }
    }
}
```

Similarly to MVC, the UIView/UIViewController hierarchy still implicitly represents the view state. However, two tasks that view controllers usually handle are factored out: the view-model transforms model values into the data needed by the views, and the coordinator manages the flow of view controllers.

Testing

With MVC, we showed an *integration testing* approach. Considering the strong integration between view controllers, view state, and views, this is the only practical testing approach.

In an integration test, we don't just test the internal logic of a component, but also the connections to other components — and sometimes we even test those components. Because an integration test contains extensive knowledge about the framework (e.g. UIKit) and isn't isolated, it can be hard to determine what causes a failing test. Additionally, integration tests are harder to write using a test-first approach, because they require extensive knowledge about all the components in the program.

MVVM changes this integrated nature by defining a clear interface — the view-model — which is explicitly decoupled from the view layer. By testing this layer in isolation (instead of testing the entire view controller and all connected views), we can minimize the number of connected components we must test simultaneously.

However, MVVM interface tests and MVC integration tests are very different in scope, structure, and detail. In general, integration tests cover more functionality, but they are more difficult to write and maintain. Additionally, because setting up a testing scenario is so difficult, integration tests typically test every property in the scenario in a single test rather than narrowly testing single functions per test.

In MVVM, we usually only test the view-model using interface tests, while both view controllers and views are tested using Xcode UI tests or through human-driven tests. This means that any logic in the view controller isn't tested using interface tests. In addition, if we misunderstand how UIKit (or other APIs used in the view or view controller) function, this won't be covered in the interface tests.

Accordingly, MVVM usually includes rules that view controller code must be very simple (to the point of being mindless). Additionally, where possible, the view controller must use library-provided bindings. With these rules, there is ideally no need to test the view controller because it doesn't contain any of our own logic.

Exactly how much logic remains in the view controller ultimately rests in the programmer's hands. RxSwift (and other reactive programming frameworks) are popular in iOS MVVM implementations because they come with platform-specific binding libraries (for example, RxCocoa is the companion to RxSwift, providing bindings to UIKit). These binding libraries remove a lot of code from view controllers, keeping closer to the ideal of MVVM.

Test Setup

The setup for view-model interface tests uses the same test store construction used in the MVC integration tests, but it doesn't need to construct a view tree:

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

   (store, childFolder1, childFolder2) = constructTestingStore()

   viewModel = FolderViewModel(initialFolder: childFolder1)

   viewModel.folderContents.subscribe(onNext: { [weak self] in
      self?.contentsObserved.append($0)
   }).disposed(by: disposeBag)
   viewModel.navigationTitle.subscribe(onNext: { [weak self] in
      self?.titleObserved.append($0)
   }).disposed(by: disposeBag)
}
```

The above code constructs the testing store, including references to some of the contained folders; constructs a view-model for one of the contained folders; and then immediately subscribes to two commonly used observables on the view-model. This setup code shows a classic pattern in interface testing: construct the inputs, construct the interface passing the inputs, and read the outputs from the interface.

Presentation Tests

Testing observables follows a common pattern: for one exposed observable on the view-model, test initial conditions, perform an action, and test subsequent conditions (optionally performing further actions and testing further conditions). The test for the navigation title is an example of this pattern:

```
// FolderViewControllerTests
func testNavigationTitle() {
   // Test initial conditions
   guard titleObserved.count == 1 else { XCTFail(); return }
   XCTAssertEqual(titleObserved[0], "Child 1")

   // Perform an action
   childFolder1.setName("Another name")

   // Test subsequent conditions
   XCTAssertEqual(titleObserved.count, 2)
   XCTAssertEqual(titleObserved[1], "Another name")
   // ... further actions omitted
}
```

In contrast, the MVC integration tests did not have a test specific to the navigation title. Rather, the navigation title was tested as part of the testRootFolderStartupConfiguration and testChildFolderConfigurationAndLayout tests, which attempted to test all presented properties of their respective folder views.

Action Tests

The only other type of tests are view-model action tests. These have a pattern similar to observable tests: test initial conditions, perform an action, and test subsequent conditions. Testing folder deletion is an example of this:

```
// FolderViewControllerTests
func testFolderDelete() {
   // Test initial conditions
   XCTAssertEqual(contentsObserved.count, 1)

   // Perform an action
   viewModel.deleteItem(childFolder2)

   // Test subsequent conditions
   XCTAssertEqual(contentsObserved.count, 2)
   let sections = contentsObserved[1]
   XCTAssertEqual(sections.count, 1)
   XCTAssertEqual(sections[0].items.first?.uuid, uuid4)
}
```

Other than the fact that the action here is performed on the view-model and not the store, this test has the same structure as the previous test.

Interface Tests vs. Integration Tests

Whereas most of the tests in MVC integration testing had a different structure — following the scenario being tested — the tests in view-model interface testing are more uniform. View integration tests require extensive knowledge about both how the view hierarchy works and the handling of view effects that occur asynchronously. Meanwhile, view-model tests only require knowledge of our own view-model and its interface and rarely demand asynchronous testing.

For these reasons, view-model interface tests are widely preferred to MVC integration tests. That said, integration tests cover more than interface tests. For example, in an integration test, we can verify that the view-model is correctly bound to the view, and that view actions are correctly sent to the view-model or model. Likewise, in an integration test, we can test the behavior of the view controller that doesn't depend on the view-model (for example: view layout or interacting with other components).

The gap between the view-model and the views puts stronger pressure on apps to include Xcode UI tests. Xcode UI tests overlap significantly with code-level integration tests, so you might opt to omit them if you have thorough integration tests. However, if you're relying on view-model interface tests to test your application logic, the pressure to also include UI tests is much higher.

Discussion

At first glance, MVVM-C looks like a more complex pattern than Cocoa MVC, as the view-model adds another layer to manage. At the implementation level though, the code can become simpler if you stick to the pattern consistently. Alas, simpler doesn't necessarily mean easier; until you're familiar with common transformations, reactive code can be difficult to write and frustrating to debug. On the bright side, the carefully structured data pipelines are often less prone to error and easier to maintain in the long run.

By moving model observing and other presentation and interaction logic into an isolated class structured around data flow, MVVM addresses the problems associated with the unregulated interaction of state in MVC view controllers. Since this is the most significant problem in MVC, exacerbated when Cocoa view controllers grow overlarge, this change goes a long way toward mitigating the massive view controller problem in MVC. But there are other ways that view controllers (and view-models) can grow large, so refactoring is still necessary on an ongoing basis.

Another problem commonly encountered in MVC — the model and view getting out of sync because the observer pattern isn't applied strictly — is addressed by reactive bindings that bind view-model properties to view properties. Bindings solve this because they unify the code paths responsible for the initial configuration of views and their subsequent updates.

The introduction of reactive programming isn't without its drawbacks though: it is one of the biggest obstacles to understanding and writing an MVVM app like the Recordings application. Reactive programming frameworks come with a steep learning curve, and it takes a while to adjust to this style of programming. While the idea of reactive programming is conceptually elegant, reactive programming frameworks rely on highly abstract transformations and large numbers of types, misuse of which can leave your code unparseable by humans.

MVVM with Less Reactive Programming

Reactive programming is a technique with its own set of tradeoffs that are more attractive to some and less so to others. For some programmers then, the reliance of MVVM on this technique presents a disincentive to using the pattern. Therefore, in this section we will look at using the MVVM pattern with less or no use of reactive programming.

In the original MVVM-C implementation, we used reactive programming within the view-model and as a binding between the view-model's interface and the views. Now we'll explore two variants: first we'll look at a view-model without any internal reactive programming, and then we'll look at an alternative to reactive bindings for updating the views when the view-model changes.

Implementing a View-Model without Reactive Programming

For this variant, we'll refactor the folder view-model to function without any internal use of reactive programming. We want to keep the same view-model interface we previously had — including the exposed observables for the navigation title and the table view's contents. This allows us to keep using reactive bindings in the view controller to bind the view-model's outputs to the corresponding views.

In the original implementation, the exposed observables were implemented in terms of other, internal observables. For example, the navigationTitle observable was created by mapping over the folderUntilDeleted observable:

```
// FolderViewModel
var navigationTitle: Observable<String> {
  return folderUntilDeleted.map { folder in
    // ...
  }
}
```

Since we want to get rid of internal observables like folderUntilDeleted, we have to create the exposed observables differently. For the navigation title, we create a private ReplaySubject — through which we can pass in new values — and expose it to the outside world as an observable:

```
// FolderViewModel
var navigationTitle: Observable<String> { return navigationTitleSubject }
private let navigationTitleSubject = ReplaySubject<String>.create(bufferSize: 1)
```

The folderContents observable, which is used to provide the table view with its data, is refactored in the same way.

The next step is to observe the model layer for changes. Then, we react to them by sending new values into the exposed observables if the change affects the data the view-model is serving for display. We do this by observing the store's change notification, just as in the MVC implementation of the example app:

```
// FolderViewModel
```

```
init() {
   NotificationCenter.default.addObserver(self,
      selector: #selector(handleChangeNotification(_:)),
      name: Store.changedNotification, object: nil)
}
```

The initializer sets up the observation that invokes the handleChangeNotification method for every change in the underlying data. Within this method, we check whether or not the change affects the parent folder of the displayed content. If so, we have to handle two cases: either the folder was deleted or it was changed (e.g. its title). If it's the former, we send empty values into the observables. If it's the latter, we just reset the view-model's folder property:

```
// FolderViewModel
@objc func handleChangeNotification(_ notification: Notification) {
   if let f = notification.object as? Folder, f === folder {
      let reason = notification.userInfo?[Item.changeReasonKey] as? String
      if reason == Item.removed {
         navigationTitleSubject.onNext("")
         folderContentsSubject.onNext([
            AnimatableSectionModel(model: 0, items: [])
         ])
      } else {
         folder = f
      }
   }
   // ...
}
```

If the change was not a delete, we reset the folder property to trigger its property observer:

```
// FolderViewModel
@objc func handleChangeNotification(_ notification: Notification) {
   // ...
   if let f = notification.userInfo?[Item.parentFolderKey] as? Folder,
      f === folder
   {
      folder = f
```

```
  }
}
```

The property observer on the folder property will send the latest values into the exposed observables:

```
// FolderViewModel
var folder: Folder! {
  didSet {
    let newTitle = folder.parent == nil ? .recordings : folder.name
    navigationTitleSubject.onNext(newTitle)
    folderContentsSubject.onNext([
        AnimatableSectionModel(model: 0, items: folder.contents)
    ])
  }
}
```

For a simple view-model like this, we can express the internal logic easily without using reactive pipelines. If the view-model is more complex — especially if it has to handle more internal state — reactive programming has more opportunities to shine by making the implementation more robust. The full code of the view-model from this section can be found on GitHub.

View-Models with Key-Value Observing

After removing Rx from the implementation of a view-model in the previous section, we now go one step further and look at how we can bind a view-model to views without using Rx's bindings at all. For this variant, we refactor the play view-model and play view controller to use KVO-based bindings.

First, we change all Rx observable properties on the play view-model to be to be observable through KVO by adding the @objc dynamic keywords:

```
// PlayViewModel
@objc dynamic var navigationTitle: String? = ""
@objc dynamic var hasRecording = false
@objc dynamic var noRecording = true
@objc dynamic var timeLabelText: String? = nil
```

```
@objc dynamic var durationLabelText: String? = nil
@objc dynamic var sliderDuration: Float = 1.0
// ...
```

Now we can use the code from the original MVC implementation of the play view controller to update these properties whenever the selected recording or the play state changes. The only difference is that we set the new values on the KVO observable properties, whereas the original implementation set the values on the views directly. For example, we used the following method in the MVC play view controller to update the player's progress views:

```
// PlayViewController (MVC variant)
func updateProgressDisplays(progress: TimeInterval, duration: TimeInterval) {
   progressLabel?.text = timeString(progress)
   durationLabel?.text = timeString(duration)
   progressSlider?.maximumValue = Float(duration)
   progressSlider?.value = Float(progress)
   updatePlayButton()
}
```

In the KVO-based play view-model, we can reuse the above code like this:

```
// PlayViewModel (KVO-based)
func updateProgressDisplays(progress: TimeInterval?, duration: TimeInterval?) {
   timeLabelText = timeString(progress ?? 0)
   durationLabelText = timeString(duration ?? 0)
   sliderDuration = Float(duration ?? 0)
   sliderProgress = Float(progress ?? 0)
   updatePlayButton()
}
```

The play view controller's task is to observe the relevant properties on the view-model using KVO and to forward their values to the respective view properties. For this, we use Swift 4's key-path-based KVO API to implement a simple bind method:

```
extension NSObjectProtocol where Self: NSObject {
   func observe<Value>(_ keyPath: KeyPath<Self, Value>,
      onChange: @escaping (Value) -> ()) -> NSKeyValueObservation
   {
      return observe(keyPath, options: [.initial, .new]) { _, change in
         guard let newValue = change.newValue else { return }
         onChange(newValue)
      }
   }
   func bind<Value, Target>(_ sourceKeyPath: KeyPath<Self, Value>,
      to target: Target,
      at targetKeyPath: ReferenceWritableKeyPath<Target, Value>)
      -> NSKeyValueObservation
   {
      return observe(sourceKeyPath) { target[keyPath: targetKeyPath] = $0 }
   }
}
```

The observe(_:onChange:) is a wrapper around the native KVO API. We can store the returned NSKeyValueObservation in a property on the view controller to couple the view controller's lifetime to the lifetime of its observations. On top of that, we've built the bind helper method, which observes a property on one object and sets the new values on another object's property automatically. Note that we've used the .initial option for the observation: this means that the observer will be called back immediately, unifying the code for setting the initial value and reacting to subsequent changes.

In the play view controller's viewDidLoad method, we use the binding helper from above to bind the view-model's properties to the views, just as we did with Rx's bindings:

```
// PlayViewController
var observations: [NSKeyValueObservation] = []

override func viewDidLoad() {
   super.viewDidLoad()
   observations = [
      viewModel.bind(\.navigationTitle, to: navigationItem, at: \.title),
      viewModel.bind(\.hasRecording, to: noRecordingLabel, at: \.isHidden),
      viewModel.bind(\.noRecording, to: activeItemElements, at: \.isHidden),
      viewModel.bind(\.timeLabelText, to: progressLabel, at: \.text),
      viewModel.bind(\.durationLabelText, to: durationLabel, at: \.text),
      viewModel.bind(\.sliderDuration, to: progressSlider, at: \.maximumValue),
      viewModel.bind(\.sliderProgress, to: progressSlider, at: \.value),
      viewModel.observe(\.playButtonTitle) { [playButton] in
```

```
        playButton!.setTitle($0, for: .normal)
      },
      viewModel.bind(\.nameText, to: nameTextField, at: \.text)
    ]
}
```

This KVO-based version of the play view-model is almost identical to the play view controller from the MVC variant of the app. The main difference is that the view-model sets the display data on observable properties, whereas the original play view controller sets it directly on the views.

In this approach, we still gain the isolated testability of view-models without depending on a large reactive framework. However, we lose a lot of convenience provided by Rx's bindings, e.g. when working with table views. In the Rx-based MVVM implementation, we used Rx's data source extensions to drive table view animations without any code of our own. With simple KVO-based bindings, we would have to come up with our own mechanism to drive fine-grained table view updates.

Lessons to Be Learned

Even if your code base doesn't follow the MVVM pattern, there are insights that can be applied to Cocoa MVC as well.

Introducing Additional Layers

MVVM offers lessons about abstractions that can be applied to any code base. In particular, we can construct data pipelines, which transform abstract data (from the model) into specific data (for the view). In MVVM, the view-model is a single pipeline between the model and the view controller. However, we can apply this pattern to other parts of our program as well. Here are some examples of pipelines between different components:

→ *App-model* — takes model data (e.g. whether or not there are any saved user credentials) and combines it with system service information (e.g. whether or not the network is available) and offers this information as observables to be used by other view-models. When interaction with system services is not required, an app-model might be better represented as a *settings-model* (which is not a combining or transforming model, but rather a standard model layer object).

- *Session-model* — tracks details about the current login session and might need to sit between view-models and the primary model or the other interface that handles network requests.
- *Flow-model* — a model-like version of a coordinator that models navigation state as data and can combine the navigation state with model data to provide observable model data directly to view-models.
- *Use case* — any kind of interface or model that prepares a slice of the primary model and simplifies performing actions. A use case is similar to a view-model, but it isn't tied to a single view controller and can be passed or shared between view-models or offer reusable functionality within multiple view-models. Any time an app has multiple views that display the same underlying data, we can use a common use case object to simplify fetching from the model and writing back to the model.

Most large programs eventually evolve to include abstractions like these. We suggest you do not introduce additional layers prematurely. Evaluate carefully whether the change makes your code simpler (i.e. easier to understand), less prone to casual errors, and easier to maintain. There's no point to additional abstractions if they don't improve your ability to write new code.

Coordinators

Coordinators are a pattern independent of the MVVM architecture. A coordinator can be applied to Cocoa MVC apps as well in order to alleviate view controllers from one of their responsibilities, and to decouple them. Without coordinators, a view controller usually presents other view controllers — by pushing them onto the navigation stack, by presenting them modally, etc. When using a coordinator, a view controller does not present other view controllers. Instead, it calls methods on the coordinator.

A similar separation can be achieved without introducing a separate coordinator object. You can also draw a strict separation between view controllers managing the presentation of other view controllers and view controllers managing the UI. For more details on this variation of the coordinator pattern, please refer to Dave DeLong's blog series, "A Better MVC."

Data Transformation

Another lesson to be learned from MVVM is the benefit of pulling data transformation logic out of the view controller. One responsibility of the controller layer in Cocoa MVC is to transform the model data into the display data that's needed to configure the views. Often that simply means transforming strings, numbers, or dates on a model object into their displayable forms. Even in simple cases, pulling out this code cleans up the view controller and increases testability at the same time.

The benefits become more apparent when data transformation involves more logic that's more complex than simple formatting operations, e.g. when your views rely on information of what has changed. You might have to compare or diff new data against old data or integrate data from several model objects for display. All these operations can be cleanly separated from the rest of the view management code.

Networking

5

In the previous two chapters, we examined different aspects of the MVC and MVVM-C patterns: how to construct them, how they handle changes to the model, and how they manage view state. But the example app we've built doesn't yet include a networking layer, so in this chapter, we'll discuss how networking fits into the app.

Based on the MVC implementation, we wrote two variants of our sample app that add networking support. The first variant — *controller-owned networking* — essentially removes the model layer and lets the view controllers handle network requests. The second variant — *model-owned networking* — retains the model layer of the MVC version and adds a networking layer beneath it.

The controller-owned networking version fetches its data directly from the network instead of from a local store. The data from the network is not persisted; instead, the view controller caches it in memory as view state. As such, this version of the app only works with a network connection.

Compared to the approach that shares data via the model, the controller-owned networking approach makes it difficult to share data between view controllers, since they operate largely independent of each other. New data has to be actively propagated to other dependent view controllers, whereas in model-owned networking, these view controllers instead observe changes in the model. The manual propagation makes it much more difficult to ensure data consistency across the entire app. This shortcoming is addressed in the second networked variant.

The model-owned networking version connects to the same server as the controller-owned version, but the model layer from the base MVC version remains mostly unchanged and serves as an offline cache for the data from the network. Additionally, the model now also triggers and manages network requests, using their results to update the model as needed. Changes in the model are picked up by the controllers using the observer pattern, just as in the base version without networking.

The main distiction we want to highlight in this chapter is not the online/offline divide, but rather whether networking is owned by the controller layer or by the model layer. The offline capability of the model-owned networking variant is just a byproduct of the existing model layer. In theory, the controller-owned networking version could also work offline (although it arguably would be much harder to implement correctly), or the store in the model-owned networking variant could just serve as an in-memory cache.

Both the controller-owned networking and model-owned networking versions of the app are available on GitHub. Each version includes a Mac companion app that acts as the server for the iOS app.

Networking Challenges

Regardless of which architecture is used, there are a number of unique challenges when adding network support to an app, including the following:

1. Networking adds an additional source of failure; any attempt to fetch data from the network can fail for a range of reasons. We have to plan for handling these failure cases gracefully, and often it's necessary to build additional UI to display these errors.
2. Consistently observing data over the network is considerably more difficult than observing data locally, and this often results in manually triggered refresh cycles instead.
3. Networking introduces the possibility of multiple clients updating the same data, leading to potential conflicts. Data on the network might change independently of our application. References to resources might be invalidated without notification. Two clients might update the same object, forcing the server to choose which should prevail and how the resolution of the conflict should be reported to each client.

In this book, we mostly ignore these issues, since they're not specific to any architecture. Instead, in this chapter, we focus on how networking fits into the architectural patterns discussed thus far.

Controller-Owned Networking

With controller-owned networking, the view controllers are in charge of network requests. They also own the data loaded by these requests. In turn, this means the app doesn't have a model layer: each view controller manages its own data.

Controller-owned networking is an easy "ad hoc" approach of adding networking to an app, and it gives quick results. While we'd generally caution against using this technique, it has some valid use cases (which we'll discuss in more detail later on), and it's a prevalent pattern in many code bases. Almost every developer has written code like this at one point or another, which is why we wanted to include the controller-owned networking approach in the discussion.

Fetching Initial Data

The first task for a newly presented view controller is to fetch its data. In the controller-owned networking approach, this means making a network request and configuring the views once the data comes back.

In our controller-owned networking app, each view controller only maintains an in-memory cache of previously fetched data, so we definitely have to perform a refetch when the app launches. For example, the folder view controller now has to load the contents of the folder it's supposed to display. We trigger this reload from viewDidLoad:

```
// FolderViewController
override func viewDidLoad() {
   // ...
   reload()
}
```

The reload method performs a fetch if the status of the new folder isn't already loading. To keep track of this state, we added a state property to the Folder type. This property indicates whether the contents are unloaded, currently loading, or loaded:

```
struct Folder {
   enum State {
      case unloaded
      case loaded
      case loading
   }
   var state: State
   // ...
}
```

> Contrary to the MVC version of the sample app, the controller-owned networking variant uses structs to represent folders and recordings (similar to the MVC+VS and TEA variants). We chose this approach because it emphasizes the nature of the data cached in memory: it's a momentary, local snapshot from the time of the last network request — it will never be updated, unless we make another request.

In reload, we check for the folder's state to be unloaded and make a network request to fetch the folder's contents:

```
// FolderViewController
@objc func reload() {
  guard folder.state != .loading else { return }

  folder.state = .loading
  refreshControl?.beginRefreshing()
  task = URLSession.shared.load(store!.contents(of: folder)) { result in
    self.refreshControl?.endRefreshing()
    guard case let .success(contents) = result else {
      dump(result) // TODO production error handling
      return
    }
    self.folder.contents = contents
    self.folder.state = .loaded
  }
}
```

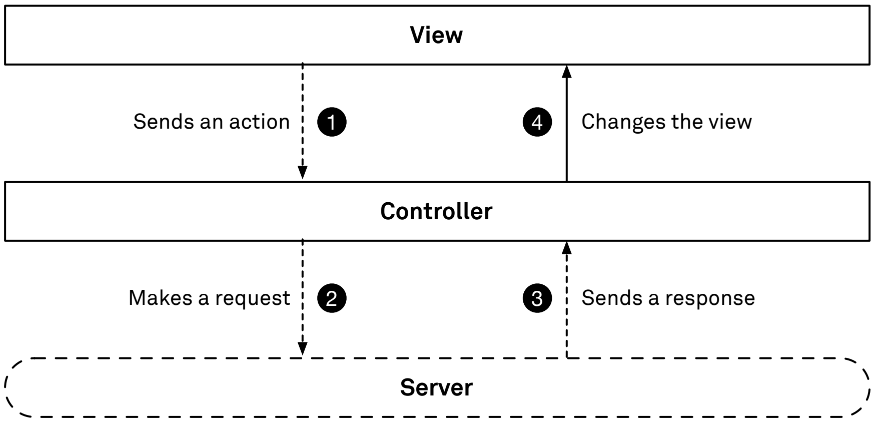
First, we set the folder's state to .loading to prevent requesting the same data again while the request is in flight. Then we perform the request, update the folder's contents, set its state to .loaded, and reload the table view. In between, we also update the state of the table view controller's refresh control.

The load method on URLSession is a custom extension that takes a resource struct, which describes the network request to be performed (we discussed this approach in more detail in the first Swift Talk episode). This wrapper around URLSession doesn't affect the event flow discussed here — using either URLSession directly or any other networking abstraction would result in the same steps.

The logic above for loading data from the network is much more complicated than the logic for loading the same data from a local store (despite our incomplete implementation, e.g. in terms of error handling). This puts additional responsibilities on view controllers, which already have to fulfill many tasks, such as managing their views and interacting with the application framework.

Making Changes

Unlike in the base MVC version, the controller-owned networking variant of our app doesn't have a model layer; we no longer have a store class that manages and persists local data. Instead, all data in our app is transient, i.e. it isn't persisted when the app quits. Rather, the task of persisting data has been entirely outsourced to the server. Therefore, making a change to the data is now an asynchronous task performed over the network. To illustrate how this affects our code, we'll look at the steps required to delete a folder:



Step 1: Table View Triggers an Action on the View Controller

This works just like in MVC: the delete button triggers the table view's data source method, tableView(_:commit:forRowAt:).

Step 2: Controller Makes a Network Request

Within this data source method, we first make sure that we're dealing with a delete event, and if so, we get the item to be deleted. The deletion code differs for folders and recordings, but we'll only follow the path for folders here:

```
override func tableView(_ tableView: UITableView,
  commit editingStyle: UITableViewCellEditingStyle,
  forRowAt indexPath: IndexPath)
{
  guard editingStyle == .delete else { return }
  let item = folder.contents[indexPath.row]
  switch item {
  case .folder:
    // Perform request...
  }
}
```

The actual deletion is a network request:

```
URLSession.shared.load(store.change(.delete, item: item)) { result in
  guard case .success = result else {
    dump(result) // TODO production error handling
    return
  }
  self.deleteItem(item: item)
}
```

Note that we don't change our cache until we receive the response from the web service. This means that the changes are transactional: only when we get a web service response that deletion succeeded do we delete the data from the cache (and from the view).

In a production app, we'd have to take care of at least two additional tasks: properly handling network errors, and giving the user some indication of the deletion being in progress. Currently, the folder just disappears after the network request has come back, which might be almost instantaneously or take a few seconds, depending on the connection. We're omitting these tasks in this example for the sake of brevity.

Steps 3 & 4: Controller Updates Its Data and the Table View

Within the network callback, the view controller's deleteItem method gets called. This method performs two tasks. It removes the deleted folder from the view controller's cached data, and it removes the corresponding row from the table view:

```
func deleteItem(item: Item) {
   guard let index = folder.contents.index(where: { $0.uuid == item.uuid })
      else { return }
   folder.contents.remove(at: index)
   tableView.deleteRows(at: [IndexPath(row: index, section: 0)], with: .automatic)
}
```

At this point, the steps for deleting an item have been completed. Aside from the asynchronous nature of this operation, the main difference to the base MVC version is that we update the table view actively in response to the user deleting a specific row. Whereas in the base version, we instructed the store to remove a particular item and the table view updated itself in response to a change notification from the store, no such mechanism exists in the controller-owned networking version.

Discussion

We stated earlier in this chapter that we don't generally recommend using the controller-owned networking approach for most apps. However, it's worthwhile to explore the tradeoffs of this approach in more detail to understand in which specific situations controller-owned networking is a viable approach and in which situations it isn't.

Since the view controllers own the data fetched from the network, this approach lacks a model layer. This means that the data owned by the view controllers is essentially view state. Locally managed view state can work fine so long as no other part of an application is dependent on it. As soon as the view state needs to be shared between view controllers, we usually create an object (outside of the view controller hierarchy) that owns this state and allows it to be changed and observed.

Applying this reasoning to the case of locally cached data from the network in a view controller, we can draw the following conclusion: if any of the data loaded from the network by a view controller needs to be shared with another part of the application, we necessarily reintroduce a model layer. If the data is only used locally, then, in principle, we could get away with building an application without a model layer, i.e. using the controller-owned networking approach.

That being said, there are a few other issues to consider. First, even though controller-owned networking might be a workable approach at the moment, we should also consider if this might change in the near future. Often, the necessity to share data across an application arises when its feature set and its complexity grows. Since a model-owned networking approach works just as well for simple apps, we might opt to use the more future-proof approach from the beginning.

Second, controller-owned networking puts a slew of new responsibilities on view controllers, including making network requests, handling the results, and handling network failures. Since view controllers usually have enough responsibilities already, we recommend not placing all this code within the view controller, but rather factoring it out into some sort of network service. This could consist of simple functions or be implemented as a web service class that handles the interaction with the server. Note that this network service is not what we'd call a model layer: it's purely a wrapper around the task of making requests and it doesn't own the resulting data.

Next, we'll take a look at a model-owned networking version of the example app and examine how it compares to the controller-owned networking version.

Model-Owned Networking

Contrary to the controller-owned networking version of the app, the model-owned networking version uses the original MVC code base without changing too much existing code. It also adds a web service component that talks directly to the store. We call it model-owned networking because the data fetched from the network is owned by the model layer.

The only added responsibility of the view controllers is to initiate the loading of data (e.g. on pull-to-refresh or when navigating to a new screen). However, the view controllers don't process the data coming back from the network as we did above in the controller-owned approach. Instead, the data gets inserted into the store. The view controllers get notified about this change the same way they do with any local change: by observing the model's change notification.

The big advantage of this approach is that it's easy to share data and communicate changes across the app. All view controllers draw their data from the store and subscribe to its change notifications. This ensures data consistency, even if multiple parts of the app display or change the same data independently.

Our model-owned networking code base also comes with some added complexity. However, this cannot be blamed on the design pattern. Rather, we decided to make the model-owned version more capable than the controller-owned one: we use the store as an offline cache to make the app work without a network connection, and we update data on the client immediately without asking the server first. Due to these enhancements, we have additional code for tracking uncommitted changes on the client and for resolving potential conflicts when submitting changes to the server.

When we look at the implementation details below, we focus on the big picture of how networking is integrated when compared to the controller-owned approach. Along the way, we only touch on the details of these added features (and complexities) as necessary, since they're not essential from the perspective of application architecture.

Fetching Initial Data

Similar to what's done in the controller-owned networking version, we fetch the initial data in the folder view controller's viewDidLoad method:

```
// FolderViewController
override func viewDidLoad() {
  // ...
  reload()
}
@objc func reload() {
  task?.cancel()
  refreshControl?.beginRefreshing()
  task = folder.loadContents { [weak self] in
    self?.refreshControl?.endRefreshing()
  }
}
```

Within the reload method, we update the refresh control's state and call loadContents on the current folder. This tells the model layer to refresh the folder's contents from the network in order to be up to date with any changes that might have happened on the server in the meantime. The model-owned networking's reload method is significantly simpler than the same method from the controller-owned implementation: we trigger the reload, but we don't have to take care of processing the request's result. This is done within the model layer, as we'll see in the following loadContents method:

```swift
extension Folder {
  @discardableResult
  func loadContents(completion: @escaping () -> ()) -> URLSessionTask? {
    let task = URLSession.shared.load(contentsResource) { [weak self] result in
      completion()
      guard case let .success(items) = result else { return }
      self?.updateContents(from: items)
    }
    return task
  }
}
```

In the code above, we perform the actual network request (using the same tiny networking abstraction as in the controller-owned variant) and process the result. This request returns an array of Items (i.e. folders or recordings), which we use to update the folder's contents. After merging the new contents from the network with the local items that have pending changes, updateContents finally sets the contents property and saves the changes:

```swift
extension Folder {
  func updateContents(from items: [Item]) {
    // ...
    contents = merged
    store?.save(self, userInfo: [Item.changeReasonKey: Item.reloaded])
  }
}
```

Just as in the base MVC version, the store sends a notification after it has saved. The folder view controller that initiated the reload picks up this notification and updates the table view through the normal model observation path, which we described in the MVC chapter.

Making Changes

Making changes to the data in the model-owned networking version works the same way as in the base MVC version. For example, to delete a folder, we take exactly the same steps: we receive the delete action through the table view's data source method, remove the folder from its parent, save the changes, and update the table view in response to the model's change notification (see Changing the Model in the MVC chapter).

This demonstrates how transparent the model-owned networking approach is to large parts of the controller layer. The controllers interact with the data from the store as if it's just local data, and the networking happens behind the scenes within the model layer.

For our model-owned networking implementation, we decided to keep the model layer from the MVC version as untouched as possible and to add a web service separate from the store. To pick up changes from the local store, this web service observes the same change notification as the view controllers:

```
final class Webservice {
  init(store: Store, remoteURL: URL) {
    // ...
    NotificationCenter.default.addObserver(self,
      selector: #selector(storeDidChange(_:)),
      name: Store.changedNotification, object: nil)
  }

  @objc func storeDidChange(_ note: Notification) {
    guard let pending = PendingItem(note) else { return }
    pendingItems.append(pending)
    processChanges()
  }
  // ...
}
```

From the notification, we extract all the information we need to commit this change to the web service, including the kind of change (e.g. an update or a delete), as well as the current attributes of the changed item. This information is stored in the PendingItem struct. We append this struct to a queue, because other pending changes might still be in flight or waiting to be committed, or we might not have connectivity in the moment. Lastly, we kick off the processing of the queue by calling processChanges.

The implementation of processChanges takes the first item from the queue, tries to commit it to the server, and processes the response. If the response is an error, we have to decide how to deal with the conflict. In the case of a success, we remove the item from the queue and send a change notification to give other components a chance to react to the new committed status of the item:

```
// WebService
func processChanges() {
   guard !processing, let pending = pendingItems.first else { return }
   processing = true
   let resource = pending.resource(remoteURL: remoteURL,
      localBaseURL: store.localBaseURL)
   URLSession.shared.load(resource) { [weak self] result in
      guard let s = self else { return }
      if case let .error(e) = result {
         // error processing ommitted...
      } else {
         s.pendingItems.removeFirst()
         // post notification ...
      }
      s.processing = false
      s.processChanges()
   }
}
```

This is an abridged version of the networking code, but the full version can be found on GitHub. In addition to the code shown above, the web service also persists the queue of pending changes locally (and loads it from disk upon initialization), in order to not lose local changes while being offline.

We only need to change the controller to improve the user experience of working with offline and online data. For example, we can add the ability to force reload data, to inform the user about errors in a graceful manner, or to indicate the offline/online status of the data onscreen. As an example, we've augmented the folder view controller's data source methods to indicate whether or not an item has pending local changes:

```
// FolderViewController
override func tableView(_ tableView: UITableView,
   cellForRowAt indexPath: IndexPath) -> UITableViewCell
```

```
{
  // ...
  let cell = tableView.dequeueReusableCell(withIdentifier: identifier,
    for: indexPath)
  cell.backgroundView?.backgroundColor = item.latestChange?.color ?? .white
  return cell
}
```

Discussion

From an architectural perspective, the major difference between the two approaches presented above — controller-owned and model-owned networking — is the ownership of the data. In controller-owned networking, the data is owned locally by view controllers, whereas in model-owned networking, the data is owned by some entity in the model layer (the Store class in our example).

Networking code in view controllers is often described as an anti-pattern. However, it doesn't have to be a problem in principle if the data from the network is only used as local view state and doesn't have to be communicated to other components. If the bulk of the networking code is properly factored out into helper functions, this approach doesn't even bloat our view controllers much. The crucial question really is this: does the data have to be shared?

As soon as data has to be shared, model-owned networking is the natural approach: since the data is owned by an entity outside of the controller layer, it can easily be shared across the app without running into data inconsistencies. This is similar to view state; if a piece of view state is only relevant to one view or one view controller, storing it locally works well. However, as soon as other components depend on the same state, we have to move it into a shared entity that can be observed. From there, it's only a small step to a shared model that is independent of the application framework.

Networking in Other Architectures

The two networking approaches described in this chapter don't apply exclusively to MVC, but also to related patterns like MVVM. The difference is that controller-owned networking would become view-model-owned networking in MVVM. Even so, the implementation is very similar.

However, the distinction between controller-owned and model-owned networking doesn't apply to patterns like MVC+VS, MAVB, or TEA. For example, the premise of MVC+VS is to represent view state explicitly as part of the model. Since the data in controller-owned networking is part of the view state, it would immediately be pushed into the model layer. Thus, the distinction between controller-owned and model-owned networking doesn't make sense in this context.

In MAVB and TEA, there are no view controllers on which a controller-owned networking approach could be implemented. Therefore, networking code naturally moves into the model layer, updating the app's state to which the views are bound (MAVB), or from which the virtual views are constructed (TEA).

Model-View-Controller+ ViewState

6

Model-View-Controller+ViewState (MVC+VS) is a variant of the MVC pattern. It maintains the same layer structure, and it uses the same platform-native mechanisms like notifications and delegation to connect components. It differs in one important aspect though: it considers view state to be part of the model layer. All view state is explicitly represented by a separate store in the model layer. The view controllers observe this view state store and update the view hierarchy accordingly. Making the view state part of the model gives us a consistent way of handling view state changes, as opposed to the ad hoc style of view state communication in MVC.

By modeling the view state as a single struct — rather than relying on the implicit view state across all view objects — the architecture makes it easier to write functionality that uses the view state. For example, serializing the entire view state becomes straightforward. We can also easily inspect the entire view state at a given point (when debugging, for example).

View State as Part of the Model

Theoretical discussions of application design often reflect the traditional role of applications as document editors. The model, as defined in typical MVC, represents a document loaded from a file on disk. The model contains the state in the application that would be saved to the document's file on disk if the user performed a save action. We will refer to this definition as the *document model*.

The list of view controllers in a navigation stack, scroll positions, tab or row selections, uncommitted text fields, and other incomplete edit operations are not normally saved to the document, and are therefore excluded from the document model. We have previously defined this document-excluded, view-related state as *view state*.

The separation of these two forms of state — the persisted document model and the transient view state — has two problems:

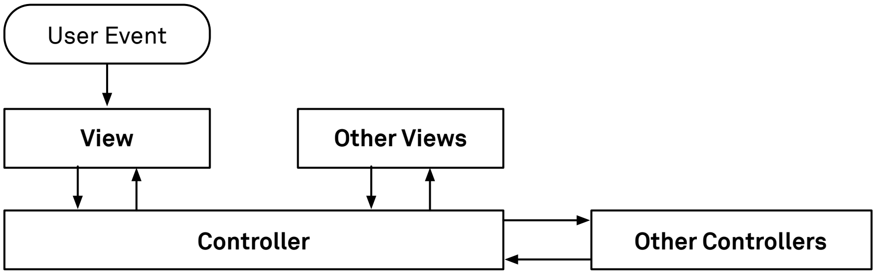
1. In modern iOS apps, the view state is expected to be persistent.
2. Non-document actions lose the benefits of model abstraction.

UIKit offers some ad hoc persistence to views (most commonly used via storyboards) to mitigate the first point, but it's the second point that can't be fixed while view state is excluded from the model.

Let’s consider a complex view action: a navigation change triggered by tapping on a table row. A whole cascade of steps can be involved in responding to this view action, such as the following:

1. Tapping a row typically updates the selection and other state on the row to provide user feedback.
2. The table view controller fetches the model object for the selected row (and might further apply state updates to the row).
3. The table view controller possibly needs to update its own local state (e.g. updating timers, animations for the transition, or views for the loss of user focus).
4. The table view controller might need to determine details of any existing view controller at the destination, so it might need to read the destination view controller’s state before updating it to the new value.

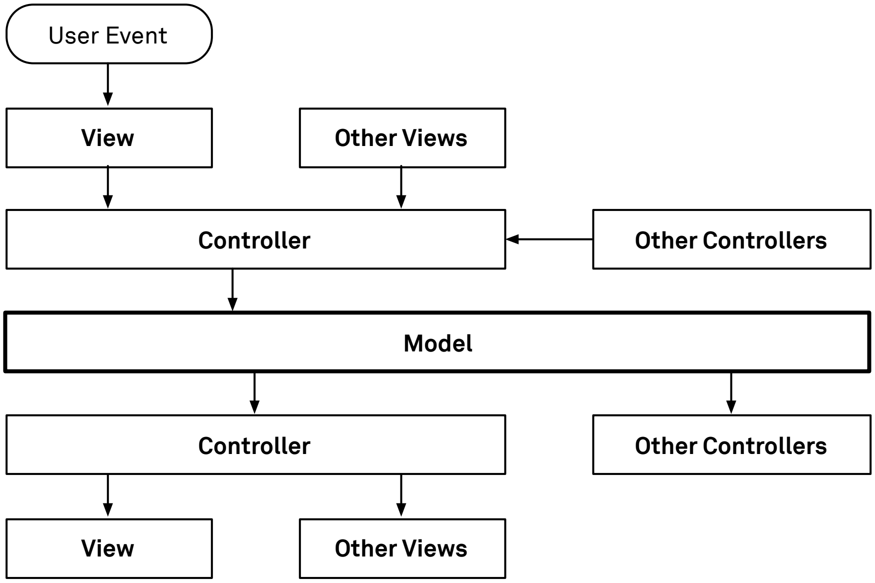
A possible data flow diagram might look like this:



The user event starts and triggers a view action, which gets forwarded to the view controller. The view controller then handles all further actions.

In MVC, data flow in view state actions has no common structure, and it often has no clear structure at all. The central controller in the action must understand the data dependencies of all other objects — including other controllers — so it can propagate the action correctly. When little view state is involved, this isn’t difficult to manage, but as the number of components grows, it becomes increasingly difficult. Getting the order of view state updates right is often a process of trial and error.

If the view action described above were handled in the same way as model actions in MVC, the diagram would look like this:



At first glance, you might think this is far more work: there are nine nodes in the graph, up from four. However, we've really just unfolded a graph that doubled back across itself into a graph that moves in just one direction — a concept called unidirectional data flow. The resulting structure keeps "How do I trigger this action?" at the top and "How do I present this state?" at the bottom.

The separation of concerns is helpful, but the greatest improvements are to data flow. The data now clearly flows top to bottom, and the central controller no longer needs to know the data dependencies of the secondary controllers (they observe the model independently).

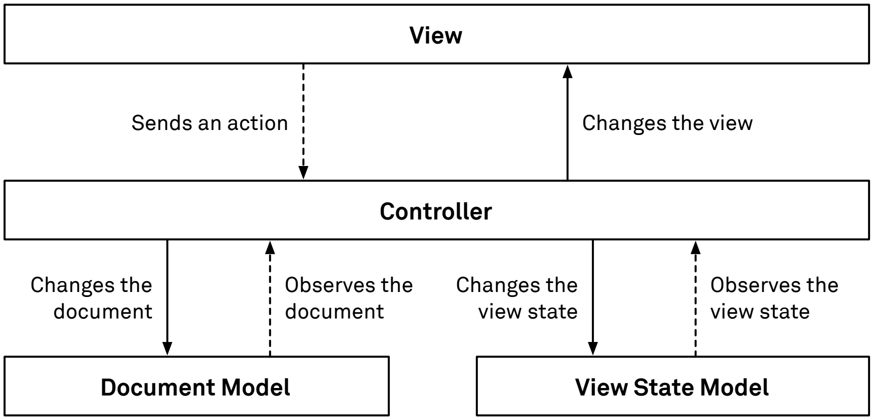
Other benefits of a model abstraction include the following:

→ The clean interface of the model permits altering the implementation without affecting the interface exposed to the rest of the program.

→ The model provides a single source of truth for the action so that refactoring and testing have a single point they can focus upon.

→ The structure scales better since the code for the bottom half of the diagram might be reused across different actions.

While MVC+VS proposes making view state part of the model layer, the view state model is kept separate from the document model because the view state has a slightly different lifecycle compared to the document model (view state is persisted using UIKit state restoration and may be purged in the event of a crash). Nevertheless, both follow the same rules:



In the view state-driven variant of the example app, the view state model and the document model are represented by the ViewStateStore and DocumentStore classes. The DocumentStore class fulfills the same role as the Store class in the MVC version of the app.

The resulting pattern has some structural similarity to TEA, with a separate view state and model, minus the need for a framework that abstracts away UIKit entirely. In MVC+VS, observing the view state and updating the views is handled by the view controllers in the same way that MVC already handles changes to the regular model.

Exploring the Implementation

Value-Typed Model Objects

The MVC+VS version of the Recordings app differs from the base MVC implementation in one important way: it uses value types for both the document model and the view state model. The `Folder` and `Recording` types are now defined as structs, as are the view state-related types like `SplitViewState` and `FolderViewState`.

We wrote the base MVC implementation using reference types (i.e. using classes) because we felt it reflected the traditional way of writing Cocoa programs, carrying on from Objective-C, where reference types dominate. However, using reference types with ownership shared between the model layer and the controller layer fails to obey the spirit of unidirectional data flow that MVC+VS aims to embody.

In unidirectional data flow, state should only be read by subscribing to it. To enforce this, our model exposes a subscription API only, with no way to read values directly. Using value types to model our data reinforces this concept — if we would have used reference types, there might have been the temptation to read from shared references to obtain new values.

Construction

Treating view state like other model state implies that we should construct the view state first, and that in an observation callback triggered by that construction, we should construct the views of the app.

However, the construction of the root view controller is an exception to this rule. Since the view controllers are the observers in this pattern, we can't observe the view state model until we have constructed at least one view controller. Because of this, the split view controller in our Recordings app is constructed unconditionally when the `Main.storyboard` file is loaded.

In our implementation, each view controller has a `context` property. The context provides a reference to both the document model and view state model. Providing it as a property avoids the need for singleton data stores or coordinators. The property must be set on the view controller before its `viewDidLoad` function is called. Usually, this occurs as part of construction (for directly instantiated view controllers) or during the prepare(for:sender:) function (for view controllers created in a segue).

The earliest that this context can be set for the root view controller is during the application delegate's application(_:didFinishLaunchingWithOptions) method:

```
func application(_ application: UIApplication,
   didFinishLaunchingWithOptions launchOptions:
      [UIApplicationLaunchOptionsKey: Any]?) -> Bool
{
   setContextOnSplitViewIfNeeded()
   // ...
}

func setContextOnSplitViewIfNeeded() {
   let splitVC = window?.rootViewController as! SplitViewController
   if splitVC.context == nil {
      splitVC.context = StoreContext(documentStore: .shared,
         viewStateStore: .shared)
   }
}
```

The split view controller has significant state — due to being the parent of the navigation stacks on both the master and detail sides — so we've opted to separate it from the application delegate and create a SplitViewController class in this version of the app. Since the split view controller also loads a master and detail view controller on startup (which display the root folder view controller and the play view controller, respectively), we must immediately propagate the context to these child view controllers too:

```
// SplitViewController
override func loadView() {
   super.loadView()
   let masterViewController = masterViewNavigationController.topViewController
      as! FolderViewController
   masterViewController.context = context.folderContext(index: 0)
   let detailViewController = (self.viewControllers.last as! UINavigationController)
      .topViewController as! PlayViewController
   detailViewController.context = context
}
```

The context for the detail view controller (the player) is the same as for the split view controller, i.e. a pair of the document store and the view state store. However, the folder view controller requires a context that also includes an index value. This allows the folder view controller to find itself in the stack of folder views represented in the view state store.

In MVC+VS, all view controllers must know how to locate themselves within the view state store. This is simple for unique view controllers like the play view controller, as there is only one playView in the view state, so it can be accessed directly. For non-unique controllers like folders in the navigation stack, an additional value must be passed in the context to uniquely identify the controller. For folder view controllers, an index is sufficient for locating them in the array of folder view controllers.

It's important to note that information passed in the context like this should only describe where the view controller will be inserted in the view hierarchy by its parent in order to locate the view controller's associated view state. The context should not include actual data (data should be stored in the view state itself).

Connecting Views to Initial Data

Once the context is set and the views are added to the window and loaded, view controllers must synchronize the state of the view hierarchy with the state described in the view state and document stores.

One of the key principles in MVC+VS is that we don't just *fetch* data from the view state or document models. Rather, we must *subscribe* to that data (so that we're guaranteed to receive future changes). In each of the view controllers, including the SplitViewController, this happens in the viewDidLoad method:

```
// SplitViewController
override func viewDidLoad() {
  // ...
  observations += context.viewStateStore.addObserver(actionType:
    SplitViewState.Action.self) { [unowned self] state, action in
    self.state = state
    self.synchronizeMasterNavigationController(action)
    self.synchronizeDetail(action)
    self.synchronizeModalAlerts(action)
  }
}
```

The addObserver method on the shared view state store is a wrapper around the standard Cocoa notification center. The actionType parameter allows us to only receive callbacks for actions we're interested in (each view state change is associated with an action describing its origin; in this case, we only want to know about view state changes originating from split view actions).

The observation callback is invoked *immediately* when we add the observer (a suggestion we described in the improvements section of the MVC chapter). The immediate callback receives the initial state value (an instance of SplitViewState), and a nil value for the action (indicating a full view state reload, rather than a user action). The SplitViewState contains a number of nested struct values specifying the view state of different components of the app:

```
struct SplitViewState: Codable {
  var folderViews: [FolderViewState]
  var playView: PlayViewState
  var recordView: RecordViewState?
  var textAlert: TextAlertState?
  // ...
}
```

Since the split view controller has a number of separate subtrees to manage, its view state observation handling is split into three synchronize... methods. These start the creation of the remaining master hierarchy, detail hierarchy, and modal hierarchy, following the structure described in the split view state. For example, the navigation stack is constructed in the synchronizeMasterNavigationController method:

```
// SplitViewController
func synchronizeMasterNavigationController(_ action: SplitViewState.Action?) {
  switch action {
  // ... non-nil cases omitted
  case nil:
    masterViewNavigationController.viewControllers =
      state.folderViews.indices.map
    { index in
      let fvc = storyboard!.instantiateViewController(
            withIdentifier: .folderIdentifier) as! FolderViewController
         fvc.context = context.folderContext(index: index)
         return fvc
      }
   }
}
```

The state.folderViews array is directly converted into the view controllers array used in the master navigation controller. This code also shows the propagation of the context from the split view to the folder view.

The split view controller doesn't need to observe the document store, as it is solely dependent on view state, but the folder view controller does observe the document store. As noted before, document state and view state are very similar, and this is true for the observation logic as well:

```
// FolderViewController
override func viewDidLoad() {
   // ...
   observations += context.documentStore.addObserver(actionType:
      DocumentStore.Action.self) { [unowned self] (store, action) in
      guard
         let folderState = self.state,
         let folder = store[folderState.folderUUID]?.folder
         else { return }
      self.folder = folder
      self.handleStoreNotification(action: action)
   }
}
```

The only tricky point here is that the callback closure uses self.state (the folder view state) to get the UUID of the folder for this view controller. This is the standard pattern in MVC+VS: the identity of any model object should be stored in the view state rather than passed to the view controller in the context object. However, this means that the observation of the document store is dependent on the current view state and must occur *after* the view state observation is initiated.

In almost all cases, the model object identifier for a view controller (the folder UUID in this case) shouldn't change during the lifetime of the view controller, but in a hypothetical scenario where you wanted this to happen, you would need to recreate the document store observation *inside* the view state store observation callback.

The model observation callback applies the model data to the view in the handleStoreNotification method:

```
// FolderViewController
func handleStoreNotification(action: DocumentStore.Action?) {
   guard let folder = self.folder else { return }
   switch action {
   // ... non-nil cases omitted
   case nil:
      title = state?.folderUUID == DocumentStore.rootFolderUUID
         ? .recordings : folder.name
      updateCachedSortedUUIDs()
      tableView.reloadData()
   }
}
```

The switch statement contains a large number of cases related to handling subsequent changes, but the nil case is there specifically for a full reset of the view — for example, when setting the initial data for the view. Since we received the self.state and self.folder values from our two observer callbacks before this point, we can apply the data from those cached values.

State Restoration

Since MVC+VS maintains the full view state in the view state model at all times, it is trivial to save and restore the view state. The ViewStateStore class contains the root of the view state's persistent data in its content property, the SplitViewState struct. This struct — and all the structs it contains — conforms to Swift's Codable protocol, so it can be deserialized from or serialized to a JSON archive at any time.

Saving and restoring the view state is triggered from the AppDelegate:

```
// AppDelegate
func application(_ application: UIApplication,
   willEncodeRestorableStateWith coder: NSCoder)
```

```
{
   guard let data = try? ViewStateStore.shared.serialized() else { return }
   if ViewStateStore.shared.enableDebugLogging {
      print("Encoding for restoration: \(String(decoding: data, as: UTF8.self))")
   }
   coder.encode(data, forKey: .viewStateKey)
}

func application(_ application: UIApplication,
   didDecodeRestorableStateWith coder: NSCoder)
{
   guard let data = coder.decodeObject(forKey: .viewStateKey) as? Data
      else { return }
   ViewStateStore.shared.reloadAndNotify(jsonData: data)
}
```

When the content property of the ViewStateStore class changes due to a restore operation, it posts a notifyingStoreReloadNotification. Every view controller that has added itself as an observer of the view state receives an observation callback. Like with construction, the associated action is nil. The view controller is expected to examine its own state and bring it up to date with the newly reloaded state — ideally changing only those properties that need changing. The code for state restoration follows the same path as the code for initial construction, shown above.

Changing the Model

When changing the document model, MVC+VS is not substantially different to MVC. In both patterns, view controllers receive view actions (via target-action or as delegates), change the model with a method call, and then observe the change through notifications. However, since our implementation of MVC+VS uses a value-type model and a different observing approach — more strictly enforcing the idea that reading from the model must be performed by *observing* the model — there are some differences.

Let's start at the folder view controller and look at how deleting an item from the folder works. In our example app, the data source of the table view is set to the folder view controller by the storyboard. To handle the tap on the delete button, the table view invokes tableView(_:commit:forRowAt:) on the data source:

```
// FolderViewController
override func tableView(_ tableView: UITableView,
   commit editingStyle: UITableViewCellEditingStyle,
   forRowAt indexPath: IndexPath)
{
   guard editingStyle == .delete, let uuid = uuidAtIndexPath(indexPath)
      else { return }
   context.documentStore.removeItem(uuid: uuid)
}
```

The store handles this call to removeItem by verifying that the item exists and has a valid parent. It then removes the item from a local copy of the parent folder and calls updateValue on the Store.content to update the store:

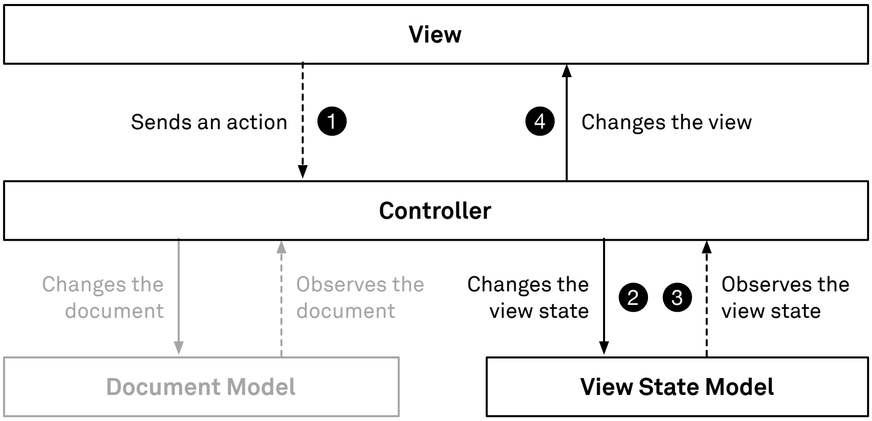
```
// DocumentStore
func removeItem(uuid: UUID) {
   guard
      let item = content[uuid],
      let parentUUID = item.parentUUID,
      var parentFolder = content[parentUUID]?.folder,
      let oldIndex = parentFolder.sortedChildUUIDs(in: self).index(of: item.uuid)
   else { return }

   parentFolder.removeChild(item.uuid, store: self)
   content.removeValue(forKey: item.uuid)
   content.updateValue(.folder(parentFolder), forKey: parentUUID)
   commitAction(Action.removed(parentUUID: parentUUID, childUUID: uuid,
      oldIndex: oldIndex))
}
```

The call to commitAction saves the store and emits a notification for the change. The FolderViewController observes this change in handleStoreNotification and deletes the corresponding row:

```
// FolderViewController
func handleStoreNotification(action: DocumentStore.Action?) {
   guard let folder = self.folder else { return }
   switch action {
   case let .removed(parentUUID, _, oldIndex)? where parentUUID == folder.uuid:
        updateCachedSortedUUIDs()
        tableView.deleteRows(at: [IndexPath(row: oldIndex, section: 0)], with: .right)
    // ...
    }
}
```

Changing the View State

In contrast to changing the model, which was very similar to the MVC approach, changing the view state is significantly different; after all, the handling of the view state is what differentiates this MVC variant. Where view state updates are typically handled by the local view or view controller in MVC, they always take a roundtrip to the view state store in MVC+VS:



We'll explore how this view state is used by looking at two different examples: toggling the play button's text in the play view controller, and pushing a new folder view controller on selection of a folder.

Example 1: Updating the Play Button

Step 1: The Button Sends an Action to the View Controller

This step is identical to the MVC version: the play button connects to the play view controller's play method using an IBAction in the storyboard.

Step 2: The View Controller Changes the View State

When the play button is tapped in MVC, the view controller immediately invokes togglePlay on the audioPlayer. We don't track this state outside of the audio player:

```
// PlayViewController
@IBAction func play() {
   audioPlayer?.togglePlay()
   updatePlayButton()
}
```

In MVC+VS, instead of directly calling through to the audio player, the play method must pass this change via the view state store. This requires a few steps, but the first is to invoke togglePlay on the view state store:

```
// PlayViewController
@IBAction func play() {
   context.viewStateStore.togglePlay()
}
```

The implementation of this togglePlay method in the view state store updates the state of the playView property — which represents the state of the play view controller — and then uses the commitAction method to send a .togglePlay action notification:

```
// ViewStateStore
func togglePlay() {
   guard var playState = content.playView.playState else { return }
   playState.isPlaying = !playState.isPlaying
   content.playView.playState = playState
   commitAction(PlayViewState.Action.togglePlay)
}
```

Step 3: The View Controller Observes View State Changes

The play view controller is subscribed to view state changes. This observation is set up in viewDidLoad, as we've seen before:

```
// PlayViewController
override func viewDidLoad() {
   // ...
   observations += context.viewStateStore.addObserver(
      actionType: PlayViewState.Action.self) { [unowned self] (state, action) in
      self.state = state.playView
      self.handleViewStateNotification(action)
   }
}
```

The closure above is called in response to any PlayViewState.Action committed in the view state. The togglePlay method committed a PlayViewState.Action.togglePlay action, and this will be handled by the corresponding case in the handleViewStateNotification method of the play view controller:

```
// PlayViewController
func handleViewStateNotification(_ action: PlayViewState.Action?) {
   switch action {
   case .togglePlay?:
      if let playState = state.playState {
         updatePlayState(playState)
      }
      audioPlayer?.togglePlay()
   // ...
   }
}
```

Step 4: The View Controller Updates the View

Within the updatePlayState method, we finally update the view hierarchy:

```
// PlayViewController
func updatePlayState(_ playState: PlayState) {
   progressLabel?.text = timeString(playState.progress)
   durationLabel?.text = timeString(playState.duration)
   progressSlider?.maximumValue = Float(playState.duration)
   progressSlider?.value = Float(playState.progress)
   playButton?.setTitle(
      playState.isPlaying ? .pause : (playState.progress > 0 ? .resume : .play),
      for: .normal)
}
```

Example 2: Pushing a Folder View Controller

Step 1: The Table View Calls Its Delegate

In MVC+VS, tapping the table view cell does not directly trigger a storyboard-connected segue like it does in MVC. Instead, we use the table view's delegate method to receive the selection event.

Step 2: The View Controller Changes the View State

Within this delegate method, we change the view state by pushing the selected folder:

```
// FolderViewController
override func tableView(_ tableView: UITableView,
   didSelectRowAt indexPath: IndexPath)
{
   guard let item = itemAtIndexPath(indexPath) else { return }
   switch item {
   case .folder(let f): context.viewStateStore.pushFolder(f.uuid)
   // ...
   }
}
```

The pushFolder implementation on the view state store is straightforward:

```
// ViewStateStore
func pushFolder(_ uuid: UUID) {
   content.folderViews.append(FolderViewState(uuid: uuid))
   commitAction(SplitViewState.Action.pushFolderView)
}
```

The view state representation of pushing a folder view controller is appending the folder's UUID to the split view state's folderViews array. Since the split view controller acts as the parent for the navigation controller containing the list of folders, this pushFolder call triggers a SplitViewState.Action notification.

Step 3: The View Controller Observes View State Changes

During initial construction of the split view controller, we added an observer — which calls the synchronizeMasterNavigationController method — to observe these actions:

```swift
// SplitViewController
override func viewDidLoad() {
  // ...
  observations += context.viewStateStore.addObserver(
    actionType: SplitViewState.Action.self) { [unowned self] state, action in
    self.state = state
    self.synchronizeMasterNavigationController(action)
    self.synchronizeDetail(action)
    self.synchronizeModalAlerts(action)
  }
}
```

Step 4: The View Controller Updates the View

Within the synchronizeMasterNavigationController method, we switch on the view state's action and react to the pushFolderView action by configuring and pushing a new folder view controller onto the navigation stack:

```swift
// SplitViewController
func synchronizeMasterNavigationController(_ action: SplitViewState.Action?) {
  switch action {
  case .pushFolderView?:
    let fvc = storyboard!.instantiateViewController(
      withIdentifier: .folderIdentifier) as! FolderViewController
    fvc.context = context.folderContext(index: state.folderViews.endIndex - 1)
    masterViewNavigationController.pushViewController(fvc, animated: true)
  // ...
  }
}
```

The new folder view controller figures out the folder it should display on its own: it inspects its position in the navigation stack and consults the view state to determine the UUID of the folder to be presented (the call to `context.viewStateStore.pushFolder` ensured this UUID is available).

Testing

View State-Driven Integration Testing

The core approach for testing an MVC+VS application is integration testing. In name, this is the same approach we used for the base MVC project, but the implementation is quite different.

In the base MVC project, we constructed a new window and built an entire view controller tree and view tree on top of this window, testing these new trees against our expectation as a single integrated unit. With MVC+VS, we only build the root view controller and provide it with the document model and view state model as its context. The view controller and view trees then build themselves in response to the view state. No matter how large or complex a view hierarchy is, building always takes the same form:

```
// FolderViewControllerTests
override func setUp() {
  super.setUp()
  documentStore = constructTestingStore()
  viewStateStore = ViewStateStore()

  let storyboard = UIStoryboard(name: "Main", bundle: Bundle.main)
  splitViewController = storyboard.instantiateInitialViewController()
    as! SplitViewController
  splitViewController.context = StoreContext(documentStore: documentStore,
    viewStateStore: viewStateStore)
  splitViewController.loadViewIfNeeded()
}
```

We construct the two stores, instantiate the root view controller from the storyboard, set the context, and load the view.

We used a default-constructed ViewStateStore above — so we'll get a default configuration of the app — but we can construct any view state we need and use that to build any possible configuration of the app. For example, here's a view state value that loads two folders — the root folder and a child folder — onto the navigation stack, selects an item in the play view controller, and presents a text alert to create a new folder in the child folder:

```
let content = SplitViewState(
  folderViews: [
    FolderViewState(uuid: DocumentStore.rootFolderUUID),
    FolderViewState(uuid: uuid1)
  ],
  playView: PlayViewState(uuid: uuid2, playState:
    PlayState(isPlaying: false, progress: 0, duration: 10)),
  textAlert: TextAlertState(text: "Text", parentUUID: uuid2, recordingUUID: nil)
)
viewStateStore.reloadAndNotify(jsonData: try! JSONEncoder().encode(content))
```

This describes the entire view state, and reloadAndNotify tells the view controller hierarchy to rebuild itself from scratch to conform to the new state. This is the strength of view state-driven programming: we can construct a new instance of the app in a specific state simply by providing the document and view state as inputs to the root view controller.

If all we want to do is verify that the view controller and view hierarchies are constructed correctly for a given combination of view state and document state, then the testing code is very similar to MVC's integration testing:

```
// FolderViewControllerTests
func testRootTableViewLayout() {
  guard let navController = splitViewController.viewControllers.first
    as? UINavigationController else { XCTFail(); return }
  guard let rootFolderViewController = navController.topViewController
    as? FolderViewController else { XCTFail(); return }
  // ...
  let firstCell = rootFolderViewController.tableView(
        rootFolderViewController.tableView,
        cellForRowAt: IndexPath(row: 0, section: 0))
    XCTAssertEqual(firstCell.textLabel!.text, "📁 Child 1")
    // ...
}
```

The first three lines are a traversal from the split view controller down to the root folder view controller. This is new: MVC already had references to the view controllers because it built them manually. For MVC+VS, we need to traverse through the automatically constructed hierarchy. However, the remaining test code — reading from the hierarchy to confirm the expected structure — is identical.

Things start to get very different from MVC testing when we look at the response to changes. In MVC, we needed to wait for animations and other asynchronous changes to complete before we could examine the view controller or view trees. This made tests difficult to write and difficult to keep reliable — especially tests on view actions that primarily affected view state.

In MVC+VS, we can separate the testing of expected view state changes in response to specific view actions from the testing of updates in the view hierarchy. More specifically, instead of waiting for view actions to perform animations and eventually affect the view tree, we can test the view state to ensure that the view actions have made the appropriate data changes (this is the *view action to view state* path). If we want to test that the view tree will be constructed to correctly reflect those view state data changes, we can write a separate test for that (the *view state to view* path).

We've already shown how to test the *view state to view* path. The previous testRootTableViewLayout test looked at the construction of the root view controller based on a certain view state. Substitute a different view state, as shown in the earlier example, and you can test any view in the program. Additionally, you can test transitions between view states by setting up the views with an initial view state, changing the view state, and testing the resulting view hierarchy against your expectations.

All that remains is to test the *view action to view state* path. Here's an example that tests selecting and navigating to a new folder:

```
// FolderViewControllerTests
func testSelectedFolder() throws {
   guard let navController = splitViewController.viewControllers.first
      as? UINavigationController else { XCTFail(); return }
   guard let rootFolderViewController = navController.topViewController
      as? FolderViewController else { XCTFail(); return }

   // Check initial conditions
   XCTAssertEqual(viewStateStore.content.folderViews.count, 1)
   // Perform change
   rootFolderViewController.tableView(rootFolderViewController.tableView,
      didSelectRowAt: IndexPath(row: 0, section: 0))

   // Check final state
   guard viewStateStore.content.folderViews.count == 2 else { XCTFail(); return }
   XCTAssertEqual(viewStateStore.content.folderViews[1].folderUUID, uuid1)
}
```

Again, the test starts with three guard statements to obtain a reference to the root folder view controller. Then we check our before state, perform the change by triggering an action on the view controller, and check the result on the view state store. For the same test case in MVC, we needed to set ourselves as the navigation controller's delegate, call performSegue, wait on an XCTestExpectation for acknowledgement by the navigation controller delegate method that the transition was complete, and then read the change from the view controller hierarchy.

In MVC+VS, by reading values from the view state, we avoid the need for delegates, asynchronous waits, and difficult-to-write-and-maintain code. We are not limited by animations. Even when we do need to read from the view controller or view hierarchies — as in *view state to view* path tests — we are reading from both view controller and view hierarchies that are not actually added to a window. This has the advantage that the tests are not subject to screen-related quirks like device orientation or screen size classes.

Interface Testing the View State Store?

The view state store represents a second model object. It might appear as though this would be a useful interface for testing, like interface testing on view-models in MVVM. However, view state is not really a testable interface, as it does not tend to have any significant logic or behaviors. Actions on the view state store tend to have setter semantics or other simple semantics like push and pop. The problem with testing actions with such simple semantics is that your tests look like this:

```
func testMyAction() {
  XCTAssert(myStore.myValue == someInitialValue)
  myStore.myValue = newValue
  XCTAssert(myStore.myValue == newValue)
}
```

This test is useless: unless you have a highly complex custom getter or setter on myValue, there is no logic tested in your program here.

This is the fundamental difference between a view state store and a view-model: the view-model is a pipeline of transformations around model objects, but the view state store is merely an observable store of plain values. In MVC+VS, the application logic — presentation and interaction — remain in the view controller. Automated testing must therefore be performed on the view controller. For automated testing, MVC+VS should keep the same integration tests that were used in MVC.

State Logging

While the view state rarely includes significant logic, it is well-suited to a different purpose: serialization. By default, the MVC+VS app logs view state to the console:

```
Restored ViewState to:
{
 "folderViews" : [
  {
   "folderUUID" : "00000000-0000-0000-0000-000000000000",
   "editing" : false,
   "scrollOffset" : 0
  },
  {
   "folderUUID" : "6950BA0E-74F5-4034-9B9F-3D3FEB48B939",
   "editing" : false,
   "scrollOffset" : 0
  }
 ],
 "playView" : {

 }
}
```

This log message indicates that when the app opened, it restored two folders on the stack (the root folder always has a UUID that is all zeros), and nothing was selected in the play view.

This type of information is useful to build test cases, and also during development for quickly understanding if an unexpected behavior is due to incorrectly set state or incorrect responses to that state. Explicit logging is also helpful to surface issues in difficult-to-observe data.

The key advantage to logging is that it can be active *before* you realize you need it. If you don't understand why previously functional behavior is non-functional, you can glance at the log while you're running the app and gain an understanding of the problem while the debugger is still active.

Time Travel Debugging

Time travel in a user interface is the idea that every time a data-driven interface changes, you can save a snapshot of all data and replay this data later to restore the user interface to a prior state. It's very similar to undo and redo, but where undo and redo affect only traditional model data, time travel also affects view state-related properties like navigation, selection, scroll position, dialogs, and more.

In most cases, time travel is a cute trick, but — unlike undo and redo — it is not a useful feature to deliver to a user. However, time travel can be useful during development for quickly verifying that an application can cleanly respond to state restoration and that all view state is correctly captured in the view state store.

In the application(_:didFinishLaunchingWithOptions: function in the app delegate, you'll see the following code:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
  launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool
{
  // ...
  #if false
     DispatchQueue.main.async {
        // Wait until *after* the main window is presented and
        // then create a new window over the top.
        self.historyViewController =
           HistoryViewController(nibName: nil, bundle: nil)
     }
  #endif
  return true
}
```

Change #if false to #if true and time travel will be enabled. A slider will appear at the bottom of the application while it is running. Drag the slider back and forth to jump through all the previous states of the application.

Note that playback state and record state are deliberately omitted from the time travel history since it causes problems to have them continually emit new states when you might be trying to drag the slider.

Discussion

Handling view state changes as discussed in this chapter requires significantly more code than handling simple view state changes in MVC. What was previously a directly applied change is now forced to be routed via the view state model interface. The upside is that we've created a consistent mechanism for communicating view state changes in our app. By enforcing the observer pattern on the global view state, mistakes that result in an out-of-sync UI are much more difficult to make.

Communication between view controllers can now occur via the view state. For example, if we want to have an additional mini player at the bottom of the folder view, we can use the same play state without having to set up a communication channel between the existing player view controller and the newly added mini player. The view state is used as the single source of truth, and both players can subscribe to it.

Some other benefits are subtler:

- → User interface state restoration requires no additional code, whereas MVC requires encoding and decoding implementations on each view controller.
- → You can log the view state (by converting it to JSON and printing it out every time it emits a notification) to observe the abstract state of the application and debug problems.
- → You can restore the state of the application, independent of `UIStateRestoring`, to repeatedly debug specific sections of your application (as discussed in the time travel debugging section above).
- → The view state model offers a place to add abstractions around navigation and flow operations for the purpose of adding features or managing complexity.

A Caveat to Tracking View State

UIKit was not designed to accurately report view state changes. You cannot always intercept view state changes before the view hierarchy is updated by UIKit, and you must update your state to match changes UIKit has already made. This is further complicated by situations where UIKit fails to clearly report *what* action occurred.

Perhaps the most difficult example of this is the handling of the back button in a navigation controller's navigation bar, particularly when the navigation controller is on the master side of a split view controller (like the folder navigation controller of the Recordings app). For most buttons, you can intercept the action, route the action via the view state model, and change the view hierarchy only in response to the notification this creates. Unfortunately, UIKit does not allow customization of navigation bar back button actions, and by the time any interceptable action is triggered, the navigation controller's stack is already changed and only limited information about what occurred is offered.

The following method — implemented in the split view controller of the Recordings app — updates the view state when the back button is tapped or a collapsed detail view is dismissed. It does this by listening to the navigation controller's delegate methods. However, these methods will also fire after we change the view hierarchy in response to a view state change (for example, when we push a new view controller, or during state restoration). We need to take care to only update the view state when the action did not originate from the view state:

```
func navigationController(_ navigationController: UINavigationController,
  didShow viewController: UIViewController, animated: Bool) {
  guard animated else { return }

  let detailOnTop = navigationController.viewControllers.last
    is UINavigationController
  if self.isCollapsed && !detailOnTop {
    // The detail view is dismissed, clear associated state
    context.viewStateStore.setPlaySelection(nil, alreadyApplied: true)
  }

  let newDepth = navigationController.viewControllers.count
    - (detailOnTop ? 1 : 0)
  if newDepth < state.folderViews.count {
    // Handle the navigation bar back button
    context.viewStateStore.popToNewDepth(newDepth, alreadyApplied: true)
  }
}
```

It's a frustrating scenario to handle, but as stated, this is one of the worst cases you're likely to encounter when handling view state in UIKit. TEA and other patterns with a declarative view layer suffer from the same problem: UIKit's view state changes independent of your view state or model.

Lessons to Be Learned

Data-Dependency Injection

Each of the application implementations in this book uses a different approach to connect each view controller to its data. The base MVC implementation sets a model object on each view controller but otherwise requires view controllers to understand the view controller hierarchy to directly talk to peers and singletons. The MVVM implementation uses a coordinator to provide context. MAVB connects all views together in a data dependency graph using reactive programming bindings. TEA runs everything through a driver that automatically provides data to functions when needed.

The implementation in this chapter uses a context property on each view controller and requires passing the context to view controllers on construction. While this isn't dramatically different from setting a model object — like in the base MVC implementation — it demonstrates a way to avoid singletons.

Our original implementation of the MVC+VS project did not use a context property. Instead, view controllers directly accessed a singleton version of the document store and the view state store. This worked well for the base app, but we were unable to write tests because the views could not be decoupled from their environment.

Ensuring a context object is correctly set before `viewDidLoad` requires more work than singletons, but it's a flexible, simple approach that can be added to any controller-based pattern, including MVC and MVVM.

Communicating Global View State

The MVC+VS pattern shown in this chapter takes the premise of view state as part of the model to the fullest extent possible in UIKit: the entire view state of the app is represented in the `ViewStateStore`. The combination of document and view state stores is the authoritative representation of the entire app, and when you want to change the view, you change the relevant store and let the views update reactively.

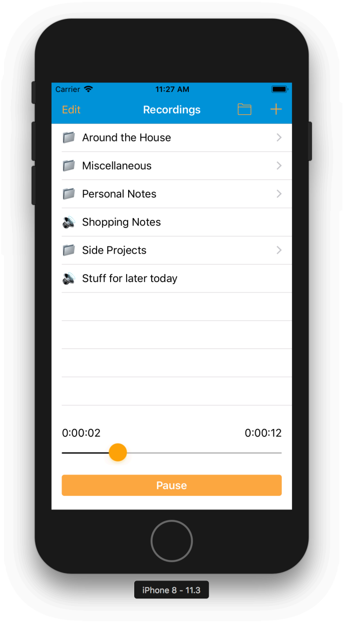
This same idea can be applied on a smaller scale to help with two common problems in MVC code bases:

1. To communicate view state between view controllers.
2. To simplify view state logic locally within a single view controller.

In this section, we'll look at an example that addresses the first point. In the next section, we'll look at an example addressing the second.

Most of the view state in the average iOS app can be managed locally — by the views themselves, or by the presenting view controller — since no other part of the app depends on this state information. However, most apps also have some view state that must be communicated between view controllers, and that's where we can borrow the idea of a view state store from MVC+VS.

As an example, we'll look at integrating a mini player at the bottom of the screen, similar to the approach used in apps like Apple Music or Spotify (this is covered in the accompanying videos as well). This scenario immediately raises a problem: before adding a mini player, the playback state can be handled locally within the play view controller, since no other part of the app depends on it. With the addition of a mini player, we'll need to communicate the playback state between the detail play view controller and the mini play view controller.



Similar to how MVC+VS lifts all view state into a view state store, we'll lift just the audio player and associated play state into a global shared player that can be observed via notifications. The player and associated playback state will behave as its own model. If we later have to deal with other global view state besides the playback state, we could choose to transform the shared player class into a more generic global view state store.

To begin, we pull the code related to the player and playback state out from the play view controller and into a shared player class:

```
final class SharedPlayer {
   var duration: TimeInterval { return audioPlayer?.duration ?? 0 }
   var progress: TimeInterval { return audioPlayer?.currentTime ?? 0 }
   var audioPlayer: Player?
```

```
    var recording: Recording? {
        didSet { updateForChangedRecording() }
    }
    // ...
}
```

Whenever the playback state or the current recording changes, we post a notification that will be observed by both the detail and the mini play view controller:

```
final class SharedPlayer {
    // ...
    static let notificationName =
        Notification.Name(rawValue: "io.objc.SharedPlayerStateChanged")
    func notify() {
        NotificationCenter.default.post(name: SharedPlayer.notificationName,
            object: self)
    }

    func updateForChangedRecording() {
        if let r = recording, let url = r.fileURL {
            audioPlayer = Player(url: url) { [weak self] time in
                self?.notify()
            }
            notify()
        } else {
            audioPlayer = nil
            notify()
        }
    }
}
```

The updateForChangedRecording method has to call notify in three different branches: in the player callback to track the playback progress, after the audio player has been initialized, and in the case that we don't have a recording. This logic could be simplified further by improving the API of the underlying Player class, but for now, it will suffice.

The play view controller directly fetches the current state from the shared player and then subscribes to notifications for subsequent updates. This is the same observer pattern as in the MVC document store:

```swift
class PlayViewController: UIViewController, UITextFieldDelegate {
  // ...
  override func viewDidLoad() {
    // ...
    updateDisplay()
    NotificationCenter.default.addObserver(self,
      selector: #selector(playerChanged(notification:)),
      name: SharedPlayer.notificationName, object: nil)
  }

  @objc func playerChanged(notification: Notification) {
    updateDisplay()
  }

  func updateDisplay() {
    updateControls() // updates time labels and playback controls
    if let r = sharedPlayer.recording {
      // ... show controls and set navigation title
    } else {
      // ... hide controls and show no recording label
    }
  }
  // ...
}
```

This could be improved by using an observing approach that sends the initial value immediately upon observing (as we used for the MVC+VS store), but we'll skip that step for the sake of brevity.

This observation of a new model — containing state that was previously treated as view state — will keep our views in sync with the player state, regardless of how many players are visible onscreen. Whenever we have view state that is shared by multiple components (e.g. selection, network activity, or user session status), we can use the same pattern.

> We used a singleton for the shared player above, since it's cumbersome to pass on the shared view state instance to all dependent view controllers from their common ancestor. However, if you use the coordinator pattern (which is

> described in the MVVM-C chapter), it becomes trivial to set the shared view state on a view controller's property or to pass it into the view controller's initializer.

Simplifying Local View State

In the previous example, we borrowed ideas from MVC+VS to create a consistent pattern for communicating view state across multiple view controllers. However, we can also apply a simplified version of MVC+VS to a single view controller in MVC by combining state properties into a single struct or enum property, observing that property, and updating the views whenever the property changes.

In the play view controller, there are a number of related properties that are always set at the same time. Whenever the recording changes, we need to set the navigation item's title, the text field's text, the progress, and the duration. Because these properties belong together, we can group them in a struct:

```
struct DisplayState {
   var name: String
   var playState: Player.PlayState
   var progress: TimeInterval
   var duration: TimeInterval
}
```

We can now add a property that holds the display state to our view controller. Whenever anything changes, we call updateViews. Note that the property is optional — either we have no selected recording, or we have a recording and all its properties available:

```
var displayState: DisplayState? = nil {
   didSet {
      updateViews()
   }
}
```

The didSet property observer is a lightweight way to have a single observer to a variable that might be changed from multiple locations — in this case, the display state.

> When we have two completely different types of state within a single view controller, we can also consider splitting up the view controller into two different view controllers: in one view controller, we might display "no recording," whereas the other view controller can only be configured with a recording and a player (it has no optional properties).

The updateViews method looks at the display state and changes the view hierarchy accordingly:

```
private func updateViews() {
  activeItemElements?.isHidden = displayState == nil
  noRecordingLabel?.isHidden = displayState != nil
  title = displayState?.name ?? ""
  nameTextField?.text = displayState?.name ?? ""
  updateProgressDisplays()
}
```

In the original MVC app, we set the view controller's title from four different places: once in the storyboard, twice in updateForChangedRecording, and once in textFieldDidEndEditing. Now, instead of changing the title, we can change the displayState property. At first, this might feel like just another level of indirection, but it simplifies reasoning about the logic of the class: to understand whether all views are configured correctly, we only have to read the updateViews method. When we want to change how the title is displayed, we can do that in a single place instead of in four places. The display state works in a way similar to a model value: all changes go through this property and are observed using a didSet property observer.

A drawback of this approach is that we are performing extra work. Now when we change just the name of a recording, we have to create a new value of DisplayState. Setting the displayState property will also update unrelated properties, such as the progress displays. We could try to look at which properties changed, or, as in MVC+VS, we could include the action that happened. Each of these options is a tradeoff between overhead and correctness, and depending on your use case, you might lean toward one or the other.

ModelAdapter-ViewBinder

7

This chapter is about the experimental architecture ModelAdapter-ViewBinder (MAVB). It's an architecture that tries to solve some of the pitfalls of MVC while still acknowledging that applications ultimately deal with mutable view hierarchies and mutable models. MAVB isolates mutation and uses clearly defined observation paths between models and views. It also provides a unified way of communicating between different parts of a program: everything is connected through one-way reactive bindings.

The MAVB pattern started as an effort to avoid view controller subclasses in MVC. What if, instead of subclassing a view controller and overriding methods, we passed closures to the view controller's constructor that both described the methods to override and delegate methods to implement? Then why not configure all the state of the view controller through construction parameters? And if we're configuring the current state, then why not use reactive programming to channel *future* state into the view controller as well?

It turns out that it's possible to pass all properties, behaviors, and observations into the controller on construction. This allows us to avoid the subclass entirely. In effect, we give the controller a long list of rules about how it should behave if specific events occur in the future — in other words, we provide it with a *declarative* system.

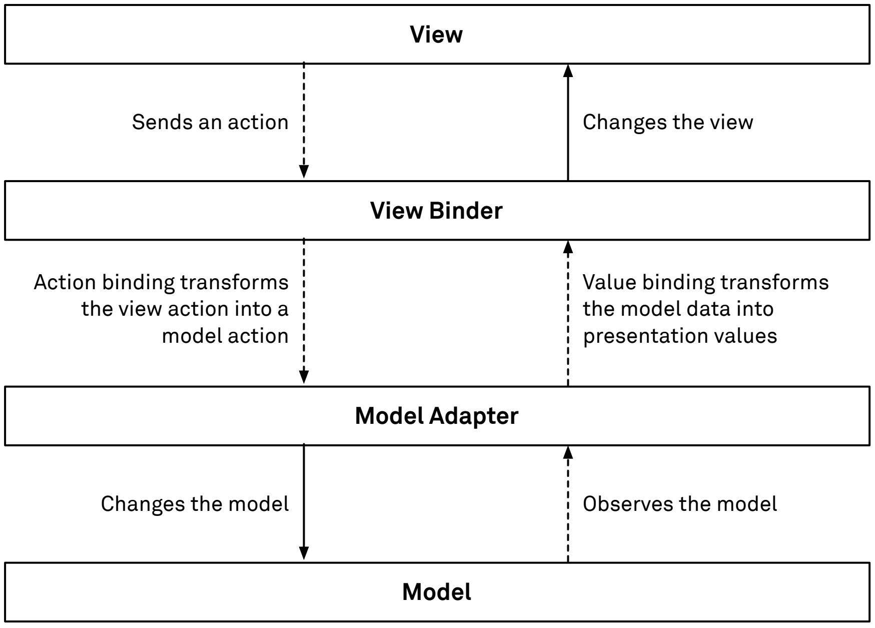
When we do this, two interesting facts emerge:

1. If we inject all behaviors into the object on construction, then the object can self-manage and doesn't need to be accessed again. In fact, it's a bad idea to access the object again, since we're only going to conflict with the self-managing behaviors.
2. View controllers exist to serve views, not themselves. If we're going to inject behavior, we should bypass the controller entirely and inject behaviors into our views.

Thus the CwlViews project began. This is a framework for constructing views with all properties and behaviors fully specified on construction so that there's never a need to access the view after construction. CwlViews *eliminates* the controller's architectural role in MVC. The MAVB pattern discussed in this chapter is a description of writing applications under the conditions created by the CwlViews framework.

In MAVB, the view layer is constructed through *view binders*. View binders are the view construction interfaces provided by CwlViews that allow views to be constructed with all of their properties and behaviors injected into them on construction. A view binder is a list of all of the closures, reactive programming bindings, and other properties that describe the view's behaviors. The view binder follows this list — collectively called the *bindings* — to automatically construct the underlying UIKit view object as needed, configuring all its properties and automatically managing all its behaviors across its lifetime.

A *model adapter* is a wrapper around a state object that forces all access to the state object to occur through reactive programming. Any changes to the underlying object must be applied via the model adapter's *signal inputs* (properties to which you can bind reactive programming observables). Correspondingly, any values or changes read from the underlying object must be read via the model adapter's output *signals* (reactive programming observables).


View
Sends an action
Changes the view
View Binder
Action binding transforms the view action into a model action
Value binding transforms the model data into presentation values
Model Adapter
Changes the model
Observes the model
Model

Where most other patterns include all of their logic within one of the layers, smaller applications in MAVB may define a significant percentage of their application logic in the reactive programming transformations of the bindings between the view binder and model adapter layers. Looking at the diagram above, this means that much of the logic lives in the arrows, rather than in the boxes.

Exploring the Implementation

A Quick Introduction to CwlViews and View Binders

Before we look at the other aspects of the design pattern, let's take a quick look at the syntax for constructing view binders, since they are an essential part of the pattern and the syntax used is a little unusual.

If we wanted a UILabel with a constant text value, we would construct a Label view binder with an immutable value for the .text binding:

```
Label(.text -- "My text label")
```

We don't usually invoke uiView() to ask the view binder to construct its view (instead, we add the view binder to a super view and this method is invoked automatically), but if we did, then the view binder would construct a UILabel and set the text property to "My text label."

The .text identifier is the *binding name*, the -- is the *static binding* operator (which indicates that the text will never change for the lifetime of the view), and "My text label" is the *binding argument*. We can think of this initialization as equivalent to the following in standard Swift parameter syntax:

```
Label(text: "My text label")
```

However, by using the binding name .text and a custom operator, we have the flexibility to switch to a different operator, which takes a mutable value instead of a constant value:

```
Label(.text <-- signalEmittingStrings)
```

The <-- operator is CwlViews' *value binding* operator. It indicates that the binding argument describes a mutable value in the form of a signal, and that data flows from the binding argument into the underlying property identified by the binding name.

In the MVVM chapter, we used RxSwift to implement reactive programming bindings, but the CwlViews framework relies instead on CwlSignal to provide reactive programming bindings. In most cases, the difference shouldn't matter, but keep in mind that in CwlSignal, the word "signal" is used as a synonym for what RxSwift calls an “observable,” and the concept of “signal input” is similar to that of a “subject” in RxSwift (a type at the head of a reactive pipeline to which you can send values).

Signal inputs are used as the binding argument with the third and final binding operator, the --> operator:

```
TextField(.textDidChange --> inputThatAcceptsStrings)
```

When the text field’s text changes, the change notification will be sent to the signal input provided as the binding argument.

```
Label(.text <-- playState.mediaState.map { timeString($0.progress) } )
```

The above line from the Recordings app creates the label showing playback progress for the player. playState.mediaState is a model adapter that contains the latest state emitted from the player. Invoking map (a reactive programming transformation function) directly on the model adapter transforms the model adapter’s default output signal. The closure takes the progress time interval from each media state struct emitted by the signal and converts it into a readable string. The resulting string-emitting signal is used to provide the text value for the label.

Connecting action bindings from the view binder to model adapters works similarly:

```
BarButtonItem(
   .barButtonSystemItem -- .add,
   .action --> Input().map { _ in .record(inParent: folder.uuid) }.bind(to: store)
),
```

This line from the Recordings app shows the action for the “+” button in the navigation bar that starts a new recording. The Input() initializer creates a new input channel (a pair consisting of the signal input and signal objects). The map function appends a transformation to the signal end of the channel. The closure ignores the void argument emitted when the button is tapped and returns a .record(inParent:) model action message. The signal end of the channel is bound to the store’s default signal input. Binding a channel consumes the signal end, so the return value is the input end of the channel. As the final step, the input end is combined with the .action binding name using the --> operator.

Scenes are constructed in MAVB by pairing a view state definition together with the construction of the scene's view controller and child views. The following is a very small example that shows all the common features:

```
struct TextState: StateContainer {
   let value: Var<String>
   init () {
      value = Var("")
   }
   var childValues: [StateContainer] { return [value] }
}

func textFieldController(_ text: TextState) -> ViewControllerConvertible {
   return ViewController(
      .view -- View(
         .layout -- .vertical(
            .view(TextField(
               .text <-- text.value,
               .didChange --> Input()
                  .map { content in content.text }
                  .bind(to: text.value)
            )),
            .space(.fillRemaining)
         )
      )
   )
}
```

There are two top-level definitions in this code sample: the view state object TextState (at the top), and the view binder construction function textFieldController (at the bottom).

The TextState contains a Var<String>. The Var type is a generic model adapter provided by the CwlViews library which offers basic setter/notifier behavior and participates in view state persistence. TextState also conforms to StateContainer so it can be part of the view state tree. We override the default implementation of childValues to return [value], which ensures that value is included during view state persistence or view state debugging.

This scene constructs a view controller with a content view, and the content view contains a single text field. The .vertical, .view, and .space identifiers used in the code sample are not binding names, but rather part of the layout description for the content view (a vertical layout with a text field view at the top and empty space filling the remainder of the view). The text field has two mutable bindings: a .didChange binding that ensures it will update the text.value model adapter when the text field content changes, and a text binding that ensures that when the text.value is updated from anywhere in the program, the text field will update accordingly.

Construction

MAVB has a very different startup approach compared to typical MVC for two reasons:

1. It doesn't have a controller layer (no UIApplicationDelegate or UIViewController subclasses).
2. It doesn't use storyboards or nib files.

Since typical Cocoa startup is usually determined by these components, taking them away leads to a very different process.

In MAVB, model adapters wrapping the top-level program state are constructed first. The store is encapsulated by a single model adapter. View state is a tree of model adapters that encapsulate a portion of the entire view hierarchy, starting with the state of the split view controller at the top level of the view hierarchy:

```
private let store = StoreAdapter(store: Store())
private let splitVar = Var<SplitState>(SplitState())
```

The store adapter (a custom model adapter) immediately wraps the underlying store model object. Similarly, the Var (a library-provided model adapter for view state that implements setting and persistence) wraps the split state, which represents the abstract state of the view hierarchy.

These objects are used as the parameters to a function that constructs the application view binder (in MAVB, the application object, like other Cocoa controller objects, is treated as a view):

```
func application(_ splitVar: Var<SplitState>, _ store: StoreAdapter)
```

```
  -> Application
{
  return Application(
    .window -- Window(
      .rootViewController <-- splitVar.map {
        splitState -> ViewControllerConvertible in
        splitView(splitState, store)
      }
    ),
    .didEnterBackground --> Input().map { .save }.bind(to: store),
    .willEncodeRestorableState -- { archiver in
      archiver.encodeLatest(from: splitVar)
    },
    .didDecodeRestorableState -- { unarchiver in
      unarchiver.decodeSend(to: splitVar)
    }
  )
}
```

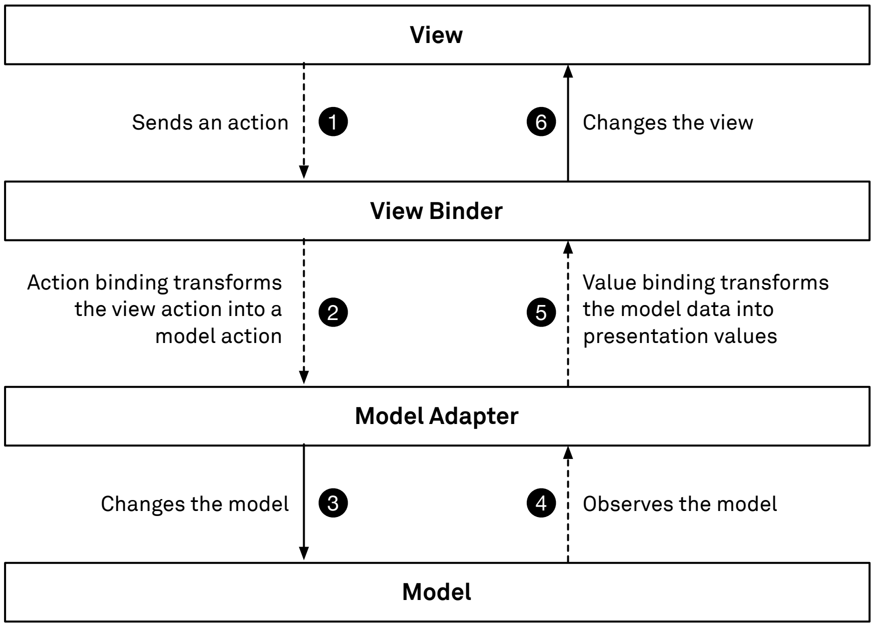
The Application view binder is configured with four bindings: .window, .didEnterBackground, .willEncodeRestorableState, and .didEncodeRestorableState.

Even though the window binding is a static binding, this immutability refers to the reference set on the application object and not to the contents of the window. When the application invokes the .didDecodeRestorableState closure during state restoration, the splitVar model adapter receives a change. In the .rootViewController binding, we observe that change using splitVar.map and replace the entire root view controller.

Changing the Model

Applying changes to the underlying model often follows a pattern called a *reducer* — a function with the signature (inout State, Action) -> Notification that updates a state object in its own isolated context. For simpler model adapters, the returned notification (which is often just a copy of the updated model state) is directly emitted through the default output signal. More complex model adapters might need to further process the notification before emitting.

In this example, we'll look at the steps involved in removing an item from a folder:


View
Sends an action
1
6
Changes the view
View Binder
Action binding transforms the view action into a model action
2
5
Value binding transforms the model data into presentation values
Model Adapter
Changes the model
3
4
Observes the model
Model

Step 1: View Actions Are Delivered to the Binder

The table view in which the user deletes the cell is constructed as follows:

```
.view -- TableView<Item>(
   // ...
   .commit --> Input()
      .compactMap { styleAndRow in styleAndRow.row.data?.uuid }
      .map { uuid in .removeItem(uuid: uuid) }
      .bind(to: store),
   // ...
)
```

Specifying the commit binding causes the internally created data source for the table view to implement the tableView(_:commit:forRowAt:) method, and any time the table view calls that method, the row description will be emitted through the binding.

Step 2: Action Bindings Transform View Actions into Model Actions

In the code above, compactMap transforms the styleAndRow parameters emitted when the tableView(_:commit:forRowAt:) method is invoked into the UUID for the row. Since this UUID is wrapped in an optional, the compactMap transformation unwraps the optional, sending only non-nil results to the next transformation stage.

The next stage is a map that transforms the non-nil UUID into a .removeItem model action message, which is then sent to the store.

Step 3: Model Adapter Applies Model Action to the Model

The StoreAdapter is primarily a wrapper around an internal FilteredAdapter. FilteredAdapter is a helper type in CwlViews that provides most of the functionality for a document model adapter. It wraps an underlying state object; provides a reducer-like closure for applying messages to that state object; and can be observed multiple times, each with an independent filter (like a slice of the underlying object or emitted notification stream), hence the name:

```
struct StoreAdapter: SignalInterface, SignalInputInterface {
  // ...
  init(store s: Store) {
    filteredAdapter = FilteredAdapter(initialState: s) {
    (store: inout Store, message: Message) -> Store.Notification in
      switch message {
      // ...
      case .removeItem(let u): return store.removeItem(uuid: u)
      // ...
      }
    }
  }
```

This shows both the initialization of the filtered adapter with the internal state object (the underlying store) and the reducer function that applies messages to the state object — in this case, we've shown the .removeItem message being handled by calling removeItem(uuid:).

Step 4: Actions on the Model Are Transformed into Signals

The call to store.removeItem(uuid: u) above immediately returns a store notification from the reducer closure. These returned notifications are emitted through a notification signal from the filtered adapter, but it's uncommon to expose that signal directly — the purpose of the filtered adapter is to produce something more customized to the needs of an observer.

The folderContentsSignal method is an example of this type of customized view into a model. The method is implemented on the store adapter and, given a folder UUID, it emits a stream of array mutations describing the contents of that folder over time, sorted for display.

The implementation relies on filteredSignal, the primary filtering output of the filtered adapter. This function takes a transformation function with access to the underlying store, the most recent notification emitted from the primary reducer function, and a state variable to enable differential communication of updates, where possible:

```swift
func folderContentsSignal(_ folderUuid: UUID) -> Signal<ArrayMutation<Item>> {
   return filteredAdapter.filteredSignal(initialValue: []) { (
      items: inout [Item],
      store: Store,
      notification: Store.Notification?,
      next: SignalNext<SetMutation<Item>>) throws in

      switch notification ?? .reload {
      case .mutation(let m) where m.values.first?.parentUuid == folderUuid:
         next.send(value: m)
      case .reload:
         if let folder = store.content[folderUuid]?.folder {
            next.send(value: .reload(folder.childUuids.compactMap {
               store.content[$0]
            }))
         } else {
            throw SignalComplete.closed
         }
      default: break
      }
   }.sortedArrayMutation(equate: { $0.uuid == $1.uuid }, compare: { l, r in
    l.name == r.name ? l.uuid.uuidString < r.uuid.uuidString : l.name < r.name
  })
}
```

In the code above, the closure signature gets in the way of the important part: the switch statement. This switch statement focuses on processing two different notifications emitted by the store:

1. .mutation notifications
2. .reload notifications

The Store.Notification.mutation notifications are set mutations describing items added, deleted, or updated from the dictionary of items in the store. We filter these to only include mutations for children of the folder we're watching. These filtered set mutations are forwarded to the next transformation without change.

When we receive the Store.Notification.reload notification (which is also sent when the notification is nil, which happens when filteredSignal is first called), we get the full set of children for the folder directly from the store, and we emit them wrapped in a SetMutation.reload.

After the processing stage created by filteredSignal, the signal is a stream of set mutations. For presentation in folder views, we want a stream of sorted array mutations, so the result is passed through another transformation stage, sortedArrayMutation, which performs that work. The resulting signal will communicate, among other changes, the removal of items from the folder contents.

Step 5: Value Bindings Transform Model Adapter Events into Presentation Values

The sequence of emitted array mutations above drives the folder's table view. In the definition of our table view, we use the folder's content signal and bind it to the table's data:

```
.view -- TableView<Item>(
  // ...
  .tableData <-- store.folderContentsSignal(folder.uuid).tableData(),
  // ...
)
```

The binding for tableData takes the array mutations in the signal and transforms them into table row mutations by calling .tableData(). When the signal emits a remove notification, it is turned into a remove mutation for the corresponding table row.

Step 6: Binder Applies Changes to the View

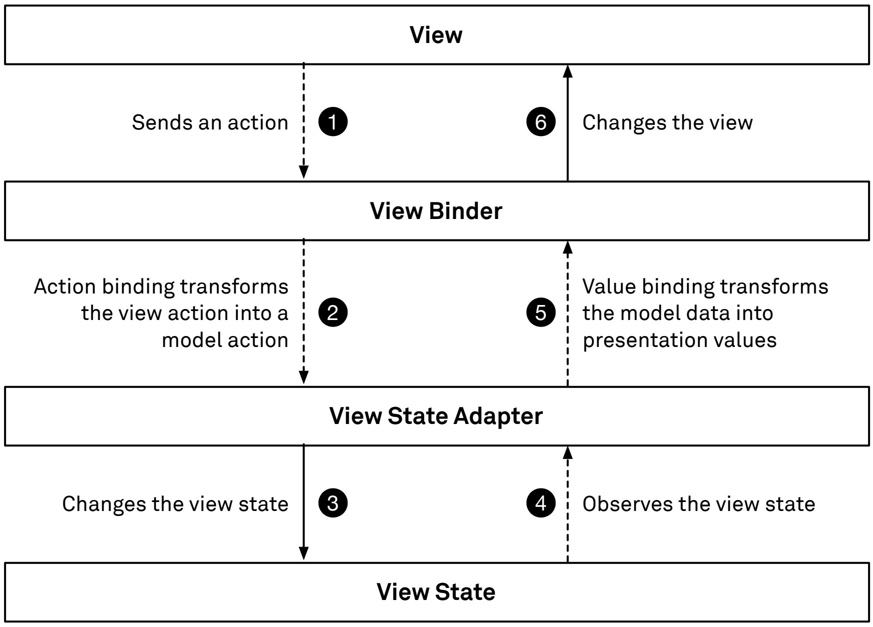
This step is done internally by the framework (in the implementation of the view binder) in the TableView.Storage.applyTableRowMutation function.

Each table row mutation value received through the binding is applied to the internally cached table data, and the kind and animation of the mutation are inspected and used to animate the table appropriately. In this case, deleteRows(at:with:) will be called with an .automatic animation effect.

Changing the View State

In this section, we'll look at the steps to change view state in MAVB, using the same two examples as in the other chapters: updating the play button's title, and pushing a new folder view controller.

Changing view state follows the same six steps we've shown for model changes above. The difference is that we're now working with view state adapters instead of model adapters:


View
Sends an action
1
6
Changes the view
View Binder
Action binding transforms the view action into a model action
2
5
Value binding transforms the model data into presentation values
View State Adapter
Changes the view state
3
4
Observes the view state
View State

A view state adapter works just like a model adapter, except that it wraps a view state value instead of a model object.

Example 1: Updating the Play Button

When the user taps the play button, it should toggle its title between playing and paused. In MAVB, this involves two cycles through the same six steps. The first cycle changes the audio player, and the second cycle changes the button.

Step 1a: The Button Tap Is Delivered to the View Binder

This step is performed by the CwlViews framework. Internally, if any actions are configured on a control (e.g. a button), a `SignalControlEventActionTarget` object will be constructed and configured as the target for the control's action in the `Control.Preparer.applyBinding` function. The object gives us a signal that emits events when the action is triggered.

Step 2a: The Primary Action Is Transformed into a View State Action

Within the action binding, we ignore the event and instead construct a .togglePlay message:

```
return Button(recordingsAppButtonBindings:
   // ...
   .action(.primaryActionTriggered) --> Input()
      .map { _ in .togglePlay }
      .bind(to: play.mediaControl)
)
```

Step 3a: The View State Adapter Updates the Media Control View State

We bind the .togglePlay message to the play.mediaControl, which is a non-persistent view state adapter for controlling the play state:

```
struct PlayState: StateContainer {
   // ...
   let mediaControl = TempVar<PlayerRecorderControl>()
   // ...
}
```

In MAVB, a view state adapter emits snapshots of its current value. In the code above, the mediaControl property is a view state adapter, and it emits values that are of type PlayerRecording. It is important to note that PlayState itself isn't a view state adapter. Instead, it is the latest value of its parent view state adapter, the playState property below:

```
struct SplitState: StateContainer {
   // ...
   let playState: Var<PlayState?>
   // ...
}
```

Both PlayState and SplitState conform to the StateContainer protocol, which provides support for serialization and debug logging. Also note that all properties of these view state values are defined using let; the only mutable parts are the view state adapters themselves.

Step 4a: The Media Control View State Change Is Observed by the Adapter

When we set the media control value using the binding .bind(to: play.mediaControl), this sends the value to the TempVar view state adapter via its input binding, which immediately emits the value to its output bindings. This reemitting work is handled internally by the TempVar implementation.

Step 5a: The Binding Transforms the Media Control Signal

In the second parameter to the audio player, we use the output signal from the TempVar:

```
AudioPlayer(
   url: play.url,
   input: play.mediaControl
      .mergeWith(AudioSession.shared.isActive.map { .audioSessionActive($0) })
   // ...
   )
)
```

Step 6a: The Media Control Is Applied to the View (AudioPlayer)

The AudioPlayer a regular class that takes bindings as input parameters and exposes only the Cancellable protocol. This makes it a de facto view binder. It might be strange to consider an audio player a view, but in MAVB, services — along with everything else outside the model — are represented as view binders.

We've gone through the six steps in the "Changing the View State" diagram and we've reached a view. However, it's not yet the button we wanted to update. To update the button label, we have to go through all six steps again.

Step 1b: The Change in Media State Is Delivered to the View Binder

Internally, the AudioPlayer class uses an AVAudioPlayerDelegate to observe the underlying AVAudioPlayer and detects the change in playback state, delivering that change in state to its output binding.

Step 2b: Action Bindings Transform View Actions into Model Actions

While the AudioPlayer output is multicast (sent to multiple destinations), we're interested in the play.mediaState destination. Since the AudioPlayer output and the play.mediaState destination contain the same type, we can use bind without a transformation:

```
AudioPlayer(
   // ...
   output: Input().multicast(
      Input()
         .ignoreElements()
         .catchError { _ in .just(nil) }
         .bind(to: split.playState),
      Input().bind(to: play.mediaState)
   )
)
```

Step 3b: The View State Adapter Updates the Media State

Because the media state is a Var, it immediately sets any received values:

```
struct PlayState: StateContainer {
   // ...
   let mediaState: Var<PlayerRecorderState>
   // ...
}
```

Step 4b: The Media State Change Is Observed by the Adapter

A Var automatically emits any new values to its output bindings.

Step 5b: The Binding Transforms the PlayState's Signal

The Button has a title binding, which transforms the media state into a string and then wraps that string. The .normal wrapper tells the button that this string is the title for the .normal control state (as opposed to .highlighted, .selected, etc.):

```
return Button(recordingsAppButtonBindings:
   .title <-- play.mediaState.map { ps in
      .normal(ps.active ? .pauseLabel : .playLabel)
   },
   // ...
)
```

Step 6b: The View Binder Changes the Button's Label

This step is performed internally by the CwlViews framework. In Button.Preparer.applyBinding, if a .title binding is provided, it is observed, and setTitle is called on the Button when an observed value is received.

Example 2: Pushing a Folder View Controller

When the user taps on a folder row inside another folder's table view, the selected folder should be presented in a new view controller.

Step 1: The Table View Selection Action Is Delivered to the View Binder

This step is performed internally by the CwlViews framework. If a row selection action is configured on a table view, an instance of TableView.Delegate will be constructed and set as the table's delegate. When this delegate receives tableView(_:didSelectRowAt:), it will send a TableRow description to the input of the configured binding.

Step 2: Action Bindings Transform View Actions into Model Actions

The .didSelectRow binding specifies folder.selection as its binding argument, so this argument will receive a value when the tableView(_:didSelectRowAt:) delegate method is invoked:

```
.view -- TableView<Item>(
   // ...
   .didSelectRow --> folder.selection,
   // ...
)
```

There is no map or other transformation specified in the binding argument; the selection property is used directly. This is possible because the view binder emits a TableRow<Item> value and the selection property is a TempVar with TableRow<Item> as its value type:

```
struct FolderState: StateContainer {
  // ..
  let selection = TempVar<TableRow<Item>>()
  // ...
}
```

Step 3: The Model Adapter Applies the Action to the View State

The folder.selection model adapter is an instance of TempVar. A TempVar can be set from the outside and can be observed, but it is not persisted. When the TempVar receives a new value via its input, it sets its state to the new value.

Steps 4 & 5: The Action Is Translated into a Notification on the Adapter

When a TempVar receives a new value, it immediately emits the new value as a notification on its output signal. A compactMap transformation is used to get the folder described by the folder.selection output signal and emits this folder only when it is non-nil. The resulting folder is transformed to a FolderState describing the next folder view to push onto the navigation state. The result of this transformation is bound to the pushInput of the split view's navigation stack:

```
.view -- TableView<Item>(
  // ...
  .cancelOnClose -- [
    folder.selection
      .compactMap { row in row.data?.folder }
      .map { folder in FolderState(folderUuid: folder.uuid) }
      .cancellableBind(to: split.navStack.pushInput),
    // ...
  ]
)
```

We use cancelOnClose together with cancellableBind to make sure the signal goes away when the table view goes away (breaking any reference cycle that the binding between the two model adapters might create).

Step 6: The NavigationController Pushes the New Folder View

The primary view of the split view controller contains the master navigation controller. The navigation controller specifies a stack parameter, which is bound to the navigation stack of the split state. The navigation stack contains the folder states, which are each transformed into folder view controllers using the folderView function:

```
private func primaryView(_ split: SplitState, _ store: StoreAdapter)
   -> NavigationControllerConvertible
{
   return NavigationController(
      .navigationBar -- navBarStyles(),
      .stack <-- split.navStack.stackMap { return folderView($0, split, store) },
      .poppedToCount --> split.navStack.poppedToCount
   )
}
```

The task of the stackMap method on the right-hand side of the binding is to transform the folder view controllers into a signal that sends stack mutations, such as push and pop. The stack mutations in the binding above are interpreted by the navigation controller's view binder. These are then translated into pushes and pops on the underlying navigation controller.

Testing

Integration Tests in MAVB

With MVC, we tested the view controller and the views simultaneously using integration tests. This type of testing is thorough but difficult to write (since it requires full knowledge of the program, app, and view frameworks), and fragile (since any change anywhere may break the test), but it can be used with any pattern.

If we wanted, we could use MVC-style integration tests in MAVB by converting the view binders to their underlying UIKit objects and testing the UIKit objects in the same way. However, the complex and fragile nature of such tests makes them undesirable unless there's a specific integration task we want to cover.

Interface Tests in MAVB

With MVVM, we tested the view-models using interface tests. Interface tests are simpler than integration tests because our tests can connect custom observers to the observables on the view-model interface rather than binding the view-model to the views. This removes the need to construct and manage views during our tests and simplifies the work involved.

We prefer interface tests like those from MVVM over integration tests like those from MVC. However, the MAVB pattern mandates that the only exposed interfaces are the model adapter interfaces (including the view state interfaces). We can always test the model via its interface — this is true for any pattern in this book — but what we really want to test is the *application* logic (the logic between the model and the views). In our sample app, the view state interfaces don't encapsulate significant view logic; view state is predominantly a series of settable values driven by view logic embedded into bindings within the view constructors, e.g.:

```
struct PlayState {
   let mediaState: Var<MediaState>
}

func constructLabel(_ play: PlayState) {
   return Label(.text <-- playState.mediaState.map { timeString($0.progress) })
}
```

To make more of the application logic testable, we could refactor these transformations and encapsulate them into the model adapters:

```
struct PlayState {
   let mediaState: Var<MediaState>
   var labelText: Signal<String> {
      return mediaState.map { state in timeString(state.progress) }
   }
}

func constructLabel(_ play: PlayState) {
   return Label(.text <-- playState. labelText)
}
```

Now the transformations have become part of our interface, and they can be tested using interface tests, just like MVVM.

A Unique Approach

The reality is that manually shifting our transformation logic into view state interfaces to create a testable interface isn't necessary in MAVB, because there is already a testable layer that doesn't need to be manually created: the view binders.

A view binder is a list of bindings (values, signals, signal inputs, and functions) in an array. We can unpack this array, inspect its contents, and use the bindings we find as the testable interface. The consumeBindings function is used to unpack the bindings from a binder. As the name implies, once bindings are consumed, the binder can no longer be used to create an actual instance. Similarly though, once an instance is created from a binder, we can no longer consume the bindings, so this kind of testing needs to be performed before objects are inserted into the active view or view controller hierarchy.

As with any interface test, we'll need to construct some input parameters to drive our tests. In the code below, we construct a sample Store:

```
func constructTestingStore() -> StoreAdapter {
   let store = Store(relativePath: "store.json", items: [
      Item(Folder(name: "Recordings", uuid: Store.rootFolderUuid,
         parentUuid: nil, childUuids: [uuid1, uuid3])),
      Item(Folder(name: "Child 1", uuid: uuid1,
         parentUuid: Store.rootFolderUuid, childUuids: [uuid2, uuid4])),
      Item(Folder(name: "Child 2", uuid: uuid2,
         parentUuid: uuid1, childUuids: [])),
      Item(Recording(name: "Recording 1", uuid: uuid3,
         parentUuid: Store.rootFolderUuid)),
      Item(Recording(name: "Recording 2", uuid: uuid4,
         parentUuid: uuid1)),
   ])
   return StoreAdapter(store: store)
}
```

Since testing bindings involves significant construction and traversing through binding structures, it is helpful to load and traverse the top level of this structure — the folderView (which returns a ViewControllerConvertible) — in the setUp method:

```
class FolderView: XCTestCase {
   var store: StoreAdapter = StoreAdapter(store: Store(relativePath: nil))
   var rootFolder: FolderState!
   var childFolder: FolderState!
   var split: SplitState!
   var rootViewController: ViewControllerConvertible!
   var childViewController: ViewControllerConvertible!

   override func setUp() {
      super.setUp()

      store = constructTestingStore()
      rootFolder = FolderState(folderUuid: Store.rootFolderUuid)
      childFolder = FolderState(folderUuid: uuid1)
      split = SplitState()
      rootViewController = folderView(rootFolder, split, store)
      childViewController = folderView(childFolder, split, store)
   }
   // ...
}
```

This setup method builds all our parameters and runs the folderView function in two different ways: passing the root folder, and passing a child folder.

All that remains for each test is to traverse through the bindings down to the desired property and test its current value:

```
func testRootFolderNavigationTitle() throws {
   let viewControllerBindings =
      try ViewController.consumeBindings(from: rootViewController)
   let title =
      try ViewController.Binding.value(for: .title, in: viewControllerBindings)
   XCTAssertEqual(title, "Recordings")
}
```

In many cases in MAVB, it is sufficient to test a single value since bindings should ensure observation. However, if you're interested in testing change logic, you can fetch the signal representation of the binding argument instead of the current value.

The following example demonstrates testing the name of the child view controller both before and after a change to the name of the folder in the store:

```
func testChildFolderNavigationTitle() throws {
   let bindings = try ViewController.consumeBindings(from: childViewController)
   let childTitle = try ViewController.Binding.signal(for: .title, in: bindings)

   var values = [String]()
   childTitle.subscribeValuesUntilEnd { values.append($0) }

   XCTAssertEqual(values.at(0), "Child 1")
   store.input.send(value: .renameItem(uuid: uuid1, newName: "New name"))
   XCTAssertEqual(values.at(1), "New name")
}
```

The final type of common test is testing a callback from a view. A callback occurs via an *action binding*. With an action binding, the binding argument will be a signal input. We can send values to this signal input and verify that the correct model state is changed.

Here's a test of the "Add folder" button's behavior. This button must set the text alert state on the split view (which will drive the presentation of a text alert via the bindings created on the split view controller):

```
func testCreateNewFolder() throws {
   // Consume from the view controller
   let vcBindings = try ViewController.consumeBindings(from: rootViewController)
   let item = try ViewController.Binding.value(for: .navigationItem, in: vcBindings)

   // Consume from the navigation item
   let itemBindings = try NavigationItem.consumeBindings(from: item)
   let rightItems = try NavigationItem.Binding.value(for: .rightBarButtonItems,
      in: itemBindings)
   guard let addItem = rightItems.value.at(1) else { XCTFail(); return }

   // Consume from the bar button item
   let addBindings = try BarButtonItem.consumeBindings(from: addItem)
   let targetAction = try BarButtonItem.Binding.argument(for: .action,
      in: addBindings)
   guard case .singleTarget(let actionInput) = targetAction
      else { XCTFail(); return }

   var values = [TextAlertState?]()
   split.textAlert.subscribeValuesUntilEnd { values.append($0) }
   XCTAssert(values.at(0)?.isNil == true)
   actionInput.send(value: ())
   XCTAssertEqual(values.at(1)??.parentUuid, Store.rootFolderUuid)
}
```

The actual test is at the bottom. We check the initial state (there shouldn't be any text alert yet), send the action (simulate a button tap), and test the final state (there should be a text alert with the root folder as the destination folder).

Comparison with View-Model Interface Testing

In most ways, the process of testing bindings consumed from a view binder is similar to that of testing observables exposed by a view-model, as both are typically concerned with observing reactive programming streams.

However, view-model interface testing suffers from a potential testing gap: relevant logic may exist in the view controller — between the view-model and the views — and evade testing. Logic in the final binding or logic in delegate implementations on the view controller may affect behavior without being noticed by tests. Additionally, view construction, layout, and other logic potentially handled at viewDidLoad may need to be excluded from the view-model because the view-model is not fully constructed at these times.

Bindings consumed from a view binder ensure that there is *no* user code between the tested bindings and the view. View binders encapsulate *all* properties of the view, including constant values and immutable construction-only properties. Since the view is constructed from the bindings and nothing else, the bindings can be considered a perfect representation of the view.

Another possibility is to change the architecture of MAVB to resemble MVVM. For example, we could move the transformations from the bindings into the model adapters or into a separate layer on top of the model adapters. This would allow us to test the transformation logic independently of the view binders.

Discussion

There are three primary advantages to MAVB: the clear structure of its declarative syntax, a unified communication model, and clearly defined connections between models and views. However, these points are simultaneously responsible for the two biggest drawbacks: the learning curve associated with the unique, abstract syntax, and the friction that complex syntax causes with debugging and editing tools.

Declarative Syntax

MAVB uses a declarative syntax to describe views and their bindings. The bindings are a set of unchangeable rules that concisely describe how specific features of the view should be constructed and react to changes. In plain UIKit, we would declare the view and its bindings imperatively, using normal control flow such as method calls. Other declarative systems we have used in this book are Auto Layout (rules describing layout of views), reactive programming (rules that describe data pipelines), and TEA.

We chose the declarative syntax for view binders because we think it is easier to read; to understand how a view is constructed and changed over time, you have to read a single list of unchangeable bindings. This is different from plain UIKit code, where you typically have to look through multiple methods that construct and change the view.

Declarative systems can greatly help readability, but they usually come with a steep learning curve: you have to learn the rules the system provides, you have to learn how the system interprets the rules, you have to learn how to debug the declarative programs, and sometimes you have to understand the performance implications of the rules as well.

Unified Communication

One of the main benefits of MAVB is a unified communication mechanism: all changes are communicated through reactive programming bindings. In MVC, it is not uncommon to use a number of different mechanisms — notifications, callbacks, method calls, delegates, target/action, and KVO — even in a small project.

Having a single communication mechanism helps when reading code and makes it possible to build reusable abstractions on top of it. It also makes it easier to see how the data flows through your app.

Relationships between Model and View

In MVC, the relationship between the model and the views is loosely defined. You can read initial state but fail to observe subsequent changes. You can fail to apply changes to the appropriate views. Even when there isn't a mistake, following the pipeline through your code from model to view can be difficult.

In MAVB, the relationship from model to view is clearly defined at the location where the view is constructed. It is easy to see at a glance what data populates a view or what effects view actions have on the model. It is much more difficult to read from a model without observing, and much harder to fail to connect an action correctly to the model.

View-models in MVVM describe a similar model-to-view relationship. But the entire model-to-view relationship is distributed between the view-model, controller, and storyboards, so the relationship cannot be inspected as easily.

Drawbacks

Declarative programming can be easier to read than imperative programming, but the effect is subjective. If you never learn the structure and rules that the system encodes, then it may remain cryptic. If you strongly prefer imperative programming (writing all the steps involved for yourself), then you might never prefer declarative programming.

The MAVB pattern described in this chapter relies on two large frameworks: CwlViews and CwlSignal. Both of these libraries are under active development and should be considered experimental rather than production ready. The CwlViews library in particular lacks significant testing in deployed environments.

Even beyond their implementation quality, these libraries present a significant learning curve. Used to its full extent, CwlViews wraps UIKit. While most properties, actions, delegate methods, and other behaviors from UIKit have an equivalent in CwlViews, the relationship is not always one-to-one, so some learning and reorientation is required. To use CwlViews, you have to make a choice to accept a layer between your own code and UIKit.

When writing code using the CwlViews syntax, Swift's error diagnostics are close to useless. When used correctly, the CwlViews syntax compiles easily and quickly, but when you make a mistake as simple as a missing comma or an incorrect type or an incorrect binding operator, Swift will usually highlight the third argument (even if the mistake is in the twentieth argument) and unhelpfully complain that you've forgotten the "bindings" parameter. Even when you understand Swift's shortcomings in this regard, finding the problem might require commenting out code until the problematic line is found (a binary search).

Due to the use of reactive programming, debugging can be much harder. LLDB is built for debugging imperative programs, but when debugging reactive programming, the debugger steps into and out of large numbers of internal reactive programming handler functions. These internal functions are uninteresting if we're trying to follow our own program's logic. It would be more helpful if the debugger would step between stages in the reactive pipeline, but that's not how the built-in debugger works. Instead, we must carefully manage breakpoints or use other strategies for debugging our code.

Using MAVB with MVC

It might not be obvious, since MAVB is quite visually different, but MAVB is interoperable with MVC. We can replace any view binder with standard Cocoa views or view controllers or use a view binder to replace a single view or view controller within a larger MVC application.

As an example, we can replace the record view in the MAVB version of the Recordings app with an MVC-style view controller subclass. A view controller subclass that accepts the same parameters as the view binder construction function could replace the record view constructor like this:

```
func recordView(_ record: RecordState, _ split: SplitState,
   _ store: StoreAdapter) -> ViewControllerConvertible
{
   return RecordViewController(record, split, store)
}
```

This drop-in replacement of a CwlViews ViewController view binder with a Cocoa UIViewController works because the .modalPresentation binding on the split view that calls this recordView function will accept any implementation of the ViewControllerConvertible protocol that both ViewController and UIViewController implement. The protocol has a single method to get a UIViewController; for UIViewController, it returns self, and for ViewController, it returns its lazily constructed underlying instance.

Then we need an implementation of init that captures all of the observables we've passed in:

```
class RecordViewController: UIViewController {
   let mediaControl: TempVar<PlayerRecorderControl>
   let store: StoreAdapter
   let recorder: AudioRecorder
   let mediaState: Signal<PlayerRecorderState>
   var endpoint: Cancellable?

   // View controller initializer takes same parameters as
   // view binder construction function
   init(_ record: RecordState, _ split: SplitState, _ store: StoreAdapter) {
      // ...
   }
```

For view controllers, constructing the view in code should happen in the loadView method. Following the example of MAVB, we should bind our signal inputs and signals to the views during this construction (rather than in viewDidLoad or the other post-construction methods typically used in MVC).

We observe the output of a signal and apply it to the view's values by using subscribe. The following example applies the label text to the progress label in the record view:

```
override func loadView() {
   let progressLabel = UILabel()
   endpoint = mediaState.subscribeValues {
      progressLabel.text = timeString($0.progress)
   }
   // ...
```

Implementing actions is a little more verbose, since we need to define a separate method to receive the action callback:

```
   // ... continued from previous example
   let stopButton = UIButton(type: .roundedRect)
   stopButton.addTarget(self, action: #selector(stop),
      for: .primaryActionTriggered)

   // further configuration, layout and assembly into a
   // single container view omitted ...

   self.view = containerView
}

@objc func stop(_ sender: Any?) {
   mediaControl.input.send(value: .stop)
   store.input.send(value: .recordingComplete)
}
```

If we would write all our views and view controllers in this way (without using view binders), the result would be an interesting hybrid of patterns: view state adapters combined with Cocoa view controllers. It's like a reactive programming version of MVC+VS.

Lessons to Be Learned

Composing Views

In UIKit, views are usually combined in a view controller or using a UIView subclass. In MAVB, we don't subclass views, but instead compose them out of existing views. To build reusable components, we can wrap the composed views in a function with the necessary parameters. For example, here is a function that builds a horizontal row with the player's current progress (on the left) and the duration of the recording (on the right):

```
func progressRow(_ playState: PlayState) -> Layout.Entity {
   return .horizontal(
      .matchedPair(
         .view(
            Label(.text <-- playState.mediaState.map { timeString($0.progress) } )
         ),
         .view(Label(
            .text <-- playState.mediaState.map { timeString($0.duration) },
            .textAlignment -- .right
         ))
      )
   )
}
```

We can also apply the same technique when we don't use MAVB. When we construct views in code, it's easy to build more complicated components by combining existing view subclasses and wrapping the construction in a function. For example, here's similar code that constructs a stack view with the two labels for displaying the recording's play progress:

```
func progressRow(progress: TimeInterval, duration: TimeInterval)
   -> UIStackView
{
   let progressLabel = UILabel()
   progressLabel.text = timeString(progress)

   let durationLabel = UILabel()
   durationLabel.text = timeString(duration)
   durationLabel.textAlignment = .right

   let stack = UIStackView(arrangedSubviews: [progressLabel, durationLabel])
   stack.axis = .horizontal
   return stack
}
```

For initial construction, the code above works well. However, for changes to the labels, it doesn't work well at all: we'd have to find the right subview of the stack view, cast it to a label, and set the text. In typical UIKit, we would wrap this code up inside a class instead of a function, because that way, we can expose settable properties for the progress and duration.

Still, we can reuse the idea from MAVB. For this to work, we need a reactive library. For example, here's the same example using RxSwift:

```
func progressRow(progress: Observable<TimeInterval>,
    duration: Observable<TimeInterval>) -> UIStackView
{
  let progressLabel = UILabel()
  progress.map(timeString).bind(to: progressLabel.rx.text)
    .disposed(by: disposeBag)

  let durationLabel = UILabel()
  duration.map(timeString).bind(to: durationLabel.rx.text)
    .disposed(by: disposeBag)
  durationLabel.textAlignment = .right

  let stack = UIStackView(arrangedSubviews: [progressLabel, durationLabel])
  stack.axis = .horizontal
  return stack
}
```

Once a reactive library is available, the technique above is very useful for building reusable components without subclassing UIView. Out of all the architectures in this book, it can be used directly with MVVM, MVC+VS, MAVB, and TEA.

Resource Ownership

MVC uses many different approaches to resource ownership, so it's easier to look at MVVM, since it uses a very consistent pattern. For example, if we wanted a text binding, we might write the following:

```
class MyViewController: UIViewController {
  @IBOutlet var textField: UITextField!
  let viewModel = ViewModel()
  var disposeBag = DisposeBag()

  override func viewDidLoad() {
    super.viewDidLoad()
    viewModel.text.bind(to: textField.rx.text).disposed(by: disposeBag)
  }
}
```

The maintenance of the dispose bag in this example isn't a huge technical burden — the pattern involved is simple and repeated enough that it becomes instinctive in MVVM with RxSwift — but it never stops being syntactic overhead, and there remain cases where bindings are established between objects with their own dispose bags. In these cases, ownership can be mismanaged by assigning to the wrong dispose bag.

Let's look at how CwlViews' view binders would handle the same text observing:

```
TextField(
  text <-- modelAdapter.text
)
```

The binding used here internally creates a `Cancellable` object — equivalent to the RxSwift disposable — but we don't see it and it doesn't require separate handling.

The solution isn't particularly complex: the binding describes a resource, and the resource is associated with the text field on construction, so the resource is tied to the text field (in this case, it's stored in an associated property using the Objective-C runtime).

The view binders provided by CwlViews demonstrate an alternative management arrangement where all resources are tied to their owners at construction. In C++, this arrangement is called *resource acquisition is initialization* (RAII). RAII is a clumsy and confusing name, but the premise is that when we construct a resource — in this case, a binding — the resource will naturally have a lifetime determined by a parent object (the view or scene). When the resource is handed to the view or scene, the resource or scene adopts the lifetime management. RAII is associated with clumsy concepts like scoped locks — since that's when the pattern sticks out — but it works best when the lifetime is tied to an object without us ever knowing.

Transparent lifetime management occurs when we construct the resource as a parameter during the parent's constructor or some other call to the owning parent. If the parent is itself a resource that is constructed as part of a call to its own parent (and that parent is a resource constructed as part of a call to another parent, and so on), then resource management is eliminated across the entire hierarchy.

It's difficult to get a deep hierarchy like this to work transparently in MVC or MVVM since view controllers have a muddied lifecycle (constructed by storyboards, and then passed through numerous lifecycle methods like viewDidLoad to finalize construction), but there are some subtle improvements we can make.

For example, consider the following protocol:

```
protocol ScopedOwner {
   var ownedObjects: [Any] { get set }
}

extension ScopedOwner {
   mutating func bind<O: ObservableType, P: ObserverType>(_ observable: O,
      to observer: P) where O.E == P.E
   {
      let disposable = observable.asObservable().bind(to: observer)
      ownedObjects.append(disposable)
   }
}
```

This would allow you to rewrite the original MVVM example as the following:

```
class MyViewController: UIViewController, ScopedOwner {
   @IBOutlet var textField: UITextField!
   let viewModel = ViewModel()
   var ownedObjects: [Any] = []

   override func viewDidLoad() {
      super.viewDidLoad()
      self.bind(viewModel.text, to: textField.rx.text)
   }
}
```

It’s not a huge difference compared to the original example, but we’ve simplified the binding line and encoded the ownership story into the entire view controller interface, reducing or eliminating the possibility of ownership mismanagement.

A Declarative View Layer

Instead of the sequentially executed statements or control flows like `if` and `for` that define imperative programming, declarative programming defines structures and rules — possibly data structures, lists of functions, or other expressions. These structures and rules are then interpreted by a system that can perform behaviors accordingly.

In the case of the MAVB version of the Recordings app, there are three separate declarative systems used:

1. View binders describe view behaviors declaratively, and the framework uses these descriptions to construct, configure, and manage the view hierarchy.
2. Bindings use reactive programming — another declarative system — where you specify how to transform the signal rather than managing control flow manually.
3. Layout is handled through a syntax that specifies horizontal and vertical elements within the layout, and then Auto Layout constraints are constructed to match the description.

Declarative programming is most helpful when the rules are simpler and offer less opportunity for mistakes than the imperative code that executes the rules. If the rules are well-chosen, they should make ambiguous, undefined, or otherwise ill-conditioned states impossible, thereby eliminating bugs by design.

In the MVVM chapter, we already showed how to apply reactive programming to simplify observing and transformation pipelines in different architectures. Reactive programming can also be applied in basic MVC, without the use of view-models, to the same end. TEA shows an alternative declarative approach that implements the observer pattern without reactive programming.

The layout syntax used in this MAVB application is a standalone single-file library, `CwlLayout.swift`. It helps by inferring the four different constraints required for each object (horizontal placement, vertical placement, horizontal size, and vertical size) by the arrangement of parameters in the function call, thus reducing the amount of code required.

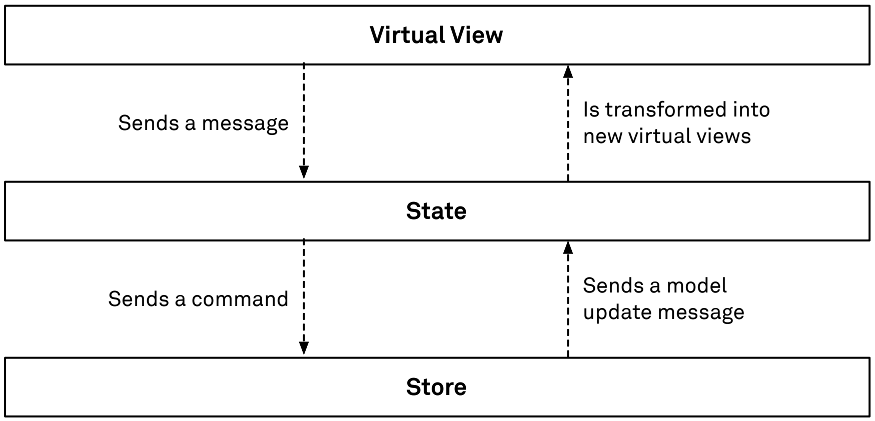
The Elm Architecture

8

The Elm Architecture (TEA) is a radical departure from MVC and other common architectural patterns on macOS and iOS. In the TEA pattern, each of the application layers we write is owned and coordinated by a supervising framework (the TEA framework). Communication between the layers, along with all interaction with UIKit, is handled by the framework, thereby shielding our code from these responsibilities. The supervising framework hides UIKit's usual imperative and object-oriented programming model, and it enables us to write our application code in a declarative, functional style.

The primary objective of using TEA on iOS is to solve the common problem of keeping the user interface consistent with the model data and the application's view state. In MVC, maintaining this consistency requires that the controller hierarchy handles numerous lifecycle and state change events (construction, view state changes, model changes) and carefully applies mutations to existing objects to reflect the total state of the application. TEA simplifies this diverse set of responsibilities down to a single pathway: whenever any state in the application changes, the application's entire state and model data are passed into a function that constructs a new, immutable, *virtual view hierarchy*. Outside of this pathway, there's no way to update views. This makes it impossible for the user interface to get out of sync with the state.

In other words: our code doesn't need to deal with applying the results of particular events to views. Instead, we describe what the view hierarchy should look like given a particular state, and the TEA framework ensures the description is followed any time state changes.


Virtual View
Sends a message
Is transformed into new virtual views
State
Sends a command
Sends a model update message
Store

In the diagram above, the app state contains all the information (view state and model data) needed to render the current user interface. The store is similar to the store of all other implementations: it takes care of loading data from disk and applying and persisting changes.

Before we continue, we should be clear about implementation quality: there is no established, production-ready TEA framework for iOS.

We've built our own partial implementation in order to show the Recordings sample app using this pattern. But while TEA is well tested as a design pattern for web apps using the Elm language, the iOS implementation of the framework used in this book should be considered an experimental proof of concept. If you're interested in using it in a production app, you will need to be prepared to adapt, extend, and maintain the framework implementation for yourself. We suggest you try it in a small side project first to understand what this might entail. Be sure to read the discussion in this chapter to learn about some of the difficulties to expect along the way.

Despite its experimental nature and the absence of production-ready frameworks, there are two reasons why we chose to include TEA:

1. TEA works so radically differently from what we're used to on Apple's platforms that it serves to widen our horizons by showing a perspective that's very different from traditional object-oriented patterns.
2. Many of the ideas from TEA have inspired other architectures and implementations; understanding their origins helps us navigate this new application architecture space. Other patterns used on iOS, such as React Native, have a significant conceptual overlap with TEA. Even architectures such as MAVB and MVC+VS take cues from TEA.

Exploring the Implementation

In this section, we'll take the TEA framework for granted and look at the application code. Consequently, none of the code described below (except the description of the startup process) interacts directly with UIKit. Application code only needs to interact with our own model layer code and the TEA framework. When we talk about view controllers and views in this section, we're not talking about instances of `UIViewController` and `UIView`, but rather of their virtual counterparts — structs and enums in the TEA framework. These structs and enums are pure values: they describe the view hierarchy, and they are later used to build the actual view hierarchy.

If you're interested in learning more about what the TEA framework does internally, please take a look at the section dedicated to this, which you'll find after the implementation details.

Construction

In our implementation, the UIApplication instance and its delegate exist outside the TEA framework, so we'll use the standard @UIApplicationMain behavior to construct them. To see the initial constuction, let's look at the top of the AppDelegate implementation:

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
  // ...
  let driver = Driver<AppState, AppState.Message>(
    AppState(rootFolder: Store.shared.rootFolder),
    update: { state, message in state.update(message) },
    view: { state in state.viewController },
    subscriptions: { state in state.subscriptions })
  // ...
}
```

The driver constructed here is the exposed interface of the TEA framework. It runs the core update loop of the application (message — update — render), it owns the model (the AppState value), and it connects all other parts of the program. This is the only location where the framework is exposed to the application; after construction, the remainder of the program executes *inside* the driver.

The driver takes four parameters:

1. The initial application state (in case of the Recordings app, a value of type AppState)
2. update — a reducer that updates the state when a new message comes in
3. view — a rendering function that creates the virtual view hierarchy from the app state

4. subscriptions — a function to get the subscriptions for an app state

The subscriptions parameter is used to dynamically compute the subscriptions when the app state changes. For example, when we're in the record state, we want to subscribe to the record progress. Later in this chapter we'll discuss subscriptions in more detail.

In application(_:didFinishLaunchingWithOptions:), the root view controller is created by using the driver's viewController property (which creates a UIViewController from the virtual view controller returned by the view parameter, above) and the result is installed at the base of the view hierarchy:

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
  func application(_ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
      [UIApplicationLaunchOptionsKey: Any]?) -> Bool
  {
    window = UIWindow(frame: UIScreen.main.bounds)
    window?.rootViewController = driver.viewController
    window?.makeKeyAndVisible()
    window?.backgroundColor = .white
    return true
  }
}
```

When the framework constructs UIKit views from virtual views, it sets up callbacks from the UIKit view objects to the driver. When a UIKit view event happens, the view object sends the corresponding message to the driver, which then changes the state by passing the message and the current state to the update function. The framework takes care of maintaining the view hierarchy for all future updates.

Connecting Views to Initial Data

TEA constructs virtual views from the app's state. When the state changes, new virtual views are constructed — there's no lasting connection. Both state and virtual views are value types (enums and structs), so they can't hold references to each other.

Populating the virtual views with their initial data follows the same path that subsequent updates do. For example, this is how the root view controller is constructed:

```
extension AppState {
  var viewController: ViewController<Message> {
    let rootView = SplitViewController<Message>(
      left: { _ in self.master },
      right: self.detail,
      collapseSecondaryViewController: playState == nil,
      popDetail: .popDetail)
    return .splitViewController(rootView, modal: recordModal)
  }
}
```

The calls to self.master, self.detail, and recordModal construct all of the potential children of the root view controller as virtual views so that the returned split view controller contains the *entire* virtual view tree. The driver will translate these into actual UIView and UIViewController instances when needed.

As an example of how the child virtual views are created, this is the definition of master:

```
extension AppState {
  var master: NavigationController<Message> {
    let viewControllers: [NavigationItem<Message>] = folders.map { folder in
      let tv: TableView<Message> = folder.tableView(
        onSelect: Message.select,
        onDelete: Message.delete)
      // ...
    }
    return NavigationController(viewControllers: viewControllers,
      back: .back, popDetail: .popDetail)
  }
}
```

The folders property on the app state is used to provide data for the view controllers in the navigation stack: we map over all folders and construct a table view for each of them. The method that constructs the table views is defined in an extension on Folder:

```
extension Folder {
  func tableView<Message>(onSelect: (Item) -> Message,
    onDelete: (Item) -> Message) -> TableView<Message>
  {
    return TableView(items: items.map { item in
      let text: String
      switch item {
      case let .folder(folder):
        text = "📁 \(folder.name)"
      case let .recording(recording):
        text = "🔊 \(recording.name)"
      }
      return TableViewCell(identity: AnyHashable(item.uuid),
        text: text, onSelect: onSelect(item), onDelete: onDelete(item))
    })
  }
}
```

The TableView type expects an array of TableViewCell values. To create the cells, we map over the folder's items and turn each one into a table view cell — this is where cell configuration happens. Back in the master property where we built the main navigation controller, we wrapped each table view in a table view controller, and then in a navigation item (providing the navigation controller with its title, bar buttons, etc.).

The entire virtual view hierarchy is constructed like this: we look at the current app state and build up the tree of virtual views that represents the state we're in.

State Restoration

In TEA, state restoration works the same as initial view construction or any other view update. As soon as the saved app state has been decoded, the driver asks the state for the virtual view hierarchy and brings it in sync with the actual view hierarchy. The state restoration hooks in the application delegate are only forwarding the actions to the driver:

```
func application(_ application: UIApplication,
  willEncodeRestorableStateWith coder: NSCoder) {
  driver.encodeRestorableState(coder)
}
```

```
func application(_ application: UIApplication,
  didDecodeRestorableStateWith coder: NSCoder) {
  driver.decodeRestorableState(coder)
}
```

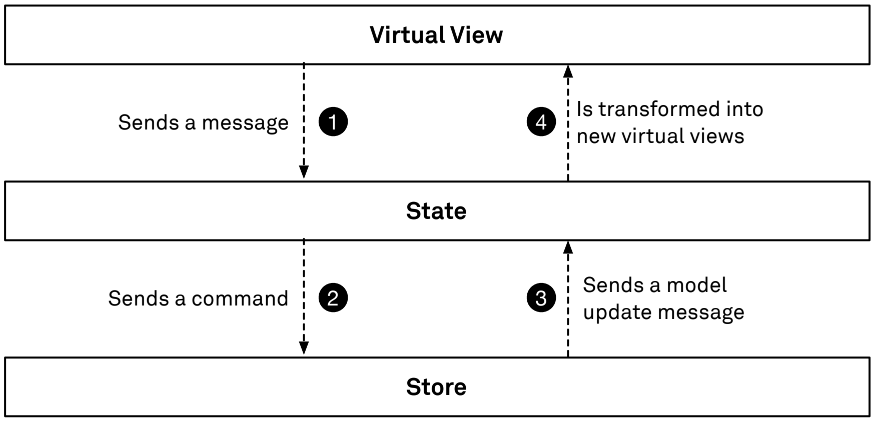
To make this work, our AppState type must conform to Codable so that the driver knows how to encode and decode it. The way TEA works, state restoration essentially comes for free. With the exception of the two delegate methods above, state restoration is independent of UIKit.

Changing the Model

In TEA, events from the underlying UIKit views are received by the driver as messages. The driver then passes these messages and the app state as parameters to the update function (which we configured as `{ state, message in state.update(message) }` in the driver's constructor).

Note that the update function does not change the data in the store directly. If it wants to change the store, it returns a command. A command is a description of an effect that should be performed — this could be a specific store update or some other task such as a network request. It's the driver that interprets these commands by executing them. For example, it might apply any needed changes to the store. Later in this chapter, we'll look at how commands work and why commands are an essential part of TEA.

The store is observed by subscriptions, so when it changes, a message is sent back to the update function updating the app's state to reflect the changed store data. From the changed app state, a new virtual view hierarchy is computed:


Virtual View
Sends a message
1
4
Is transformed into new virtual views
State
Sends a command
2
3
Sends a model update message
Store

Let's take a look at how this flow of data applies to the example of deleting an item from the folder view controller's table view.

Step 1: View Sends Message to the Update Method

The views have associated actions, also known as *messages*. In the case of a table view cell, committing a delete action results in a .delete message being sent. The .delete message is defined as part of the Message enum in the app state:

```swift
extension AppState {
  enum Message: Equatable {
    case delete(Item)
    // ...
  }
}
```

AppState.Message defines all possible messages the Recordings app can send and handle in a single place.

During setup of the table view, we specify the message that should be sent for deletion:

```swift
let tv: TableView<Message> = folder.tableView(
  onSelect: Message.select, onDelete: Message.delete)
```

The second parameter of the method constructing the table view takes a function that returns the message to be sent when the delete action is triggered. TEA configures the table view's data source to automatically send this message. The parameter passed into the function is the item in the table view.

Step 2: Update Triggers Deletion from the Store

In our application, the update method called by the driver is a mutating method on the AppState struct. It switches over the message and returns a single command in the .delete case:

```
mutating func update(_ msg: Message) -> [Command<Message>] {
  switch msg {
  case .delete(let item):
    return [Command.delete(item)]
  // ...
  }
}
```

The update method is marked as mutating because it can change the AppState. In fact, it's the only method that can change the AppState. For the .delete message, there's no state change, but instead we return a command.

Commands are descriptions of side effects (such as changing data in the store) that are interpreted by the driver. Executing the .delete command will cause the deleted item to be removed from the store. The framework provides application-independent commands, and we added application-specific commands (such as .delete) in an extension on Command.

Step 3: Update Is Notified about the Store Change

The AppState has a computed property, subscriptions, which describes what effects it wants to observe at any given time, depending on the current state. At the very least, it always includes a subscription to the store's changes:

```
extension AppState {
  var subscriptions: [Subscription<Message>] {
    var subs: [Subscription<Message>] = [
       .storeChanged(handle: { .storeChanged($0) })
    ]
    // ...
    return subs
  }
}
```

This subscription tells the following to the TEA framework: when the content of the store changes, send a storeChanged message to the app state.

When the app state's update method gets called with a .storeChanged message, it updates the state's folders array. This array contains the folders that should be on the navigation stack. Since folders are structs in this code base, we have to fetch their latest versions from the root folder, which is passed in to the .storeChanged message as an associated value:

```
mutating func update(_ msg: Message) -> [Command<Message>] {
   switch msg {
   // ...
   case .storeChanged(let root):
      folders = folders.compactMap { root.find($0) }
      // ...
      return []
   }
}
```

To refresh the folder values, we map over the existing folders array and fetch the current values from the new root folder (find performs a recursive lookup based on the folder's UUID). We use compactMap instead of map, because one of the folders might no longer be present, in which case find returns nil.

Step 4: View Is Computed from the New App State

In the previous step, we changed the app state by setting its folders property. The TEA framework notices the change of the state and recomputes the entire virtual view hierarchy. Here we only look at the part that describes the navigation stack with its folder view controllers:

```
extension AppState {
```

```swift
var master: NavigationController<Message> {
    let viewControllers: [NavigationItem<Message>] = folders.map { folder in
        let tv: TableView<Message> = folder.tableView(onSelect: Message.select,
            onDelete: Message.delete)
        return NavigationItem(title: folder.name,
            // ...
            viewController: .tableViewController(tv))
    }
    return NavigationController(viewControllers: viewControllers,
        back: .back, popDetail: .popDetail)
}
```

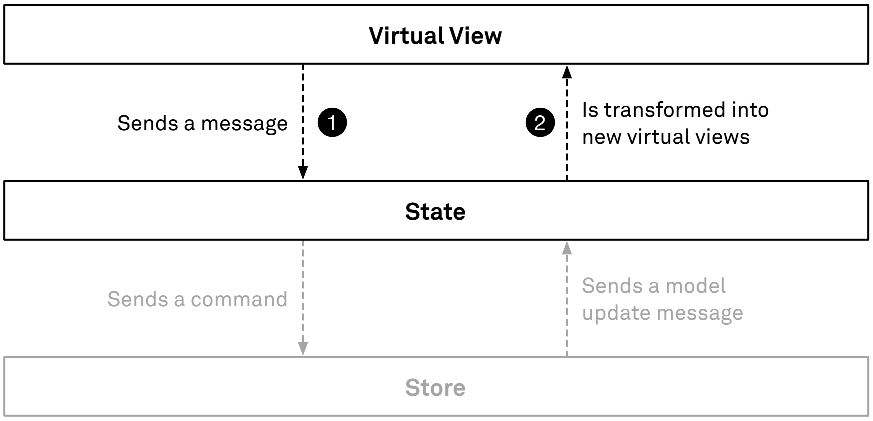
In the first line of the master property, the virtual view controllers are generated by mapping over the state's folders array. Within the map body, the virtual table view for each folder is generated by the following extension on Folder:

```swift
extension Folder {
    func tableView<Message>(onSelect: (Item) -> Message,
        onDelete: (Item) -> Message) -> TableView<Message>
    {
        return TableView(items: items.map { item in
            let text: String
            switch item {
            case let .folder(folder):
                text = "📁 \(folder.name)"
            case let .recording(recording):
                text = "🔊 \(recording.name)"
            }
            return TableViewCell(identity: AnyHashable(item.uuid), text: text,
                onSelect: onSelect(item), onDelete: onDelete(item))
        })
    }
}
```

Note that all views and view controllers generated in this code are virtual; they are structs and enums describing certain types of views and view controllers and not their actual UIKit counterparts. The driver takes this virtual view hierarchy and changes the existing UIView hierarchy. It then changes only what's different between the two hierarchies. This diffing step is smart enough to properly animate cell insertions and deletions.

Changing the View State

In a TEA app, we change the view state by sending a message to update the app's state struct, which the driver will follow with a call to generate a new virtual view hierarchy:


Virtual View
Sends a message
1
2
Is transformed into new virtual views
State
Sends a command
Sends a model update message
Store

We'll look at the same two examples for view state changes that we use in the other chapters: updating the play button's title, and pushing a folder view controller.

Example 1: Updating the Play Button

Step 1: Play Button Sends Message

In the first step, the app state's update method receives the message that's dispatched by the TEA framework upon tapping the play button. We forward this message to the player state:

```
extension AppState {
  mutating func update(_ msg: Message) -> [Command<Message>] {
    switch msg {
    case .player(let msg):
         return playState?.update(msg) ?? []
      // ...
      }
   }
}
```

The player state's update method handles the actual .togglePlay message:

```
extension PlayerState {
   mutating func update(_ action: Message) -> [Command<AppState.Message>] {
      switch action {
      case .togglePlay:
         playing = !playing
         return [Command.togglePlay(player)]
      // ...
      }
   }
}
```

We toggle the Boolean playing property and return the .togglePlay command. The driver then interprets this command and toggles the actual playing of the audio.

Step 2: The New View Hierarchy Is Computed from the State

Since the app's state has changed, the TEA framework generates a new virtual view hierarchy from the changed state. This doesn't just include the changed parts: it's a new abstract representation of the entire view/view controller hierarchy, starting from the root view controller, and continuing all the way down to our play button (and all other elements that should be onscreen).

Rather than showing the entire view construction, here's how the play view controller is computed from the player state. The player state implements a viewController property that uses an internal view property to generate the player view hierarchy:

```
extension PlayerState {
   var viewController: ViewController<Message> {
      return .viewController(view)
   }
}
```

The view property returns a virtual view — a virtual stack view to be precise — that contains the play button with the correct title:

```
extension PlayerState {
  var view: View<Message> {
    // ...
    return View<Message>.stackView(views: [
      .stackView(views: [
        // ...
        .space(height: 20),
        .button(text: playing ? .pause : .play, onTap: .togglePlay),
      ]),
      .space(width: nil, height: nil)
    ])
  }
}
```

With this piece of code, we've come full circle in our process: when we create the virtual button view, we also specify the message for tap events: .togglePlay (short for PlayerState.Message.togglePlay).

Example 2: Pushing a Folder View Controller

In the previous example — changing the title of the play button — we only changed a property of an existing view in the hierarchy. In this example — pushing a new folder view controller — we modify both the view and the view controller hierarchy. However, using the TEA framework, we take exactly the same steps in both examples.

Step 1: The Table View Sends a .selectFolder Message

When we select a folder cell in the table view, the app state's update method receives a .selectFolder message. The state change in response to this message is simple — we append the selected folder to the state's array of folders:

```
extension AppState {
  mutating func update(_ msg: Message) -> [Command<Message>] {
    switch msg {
    case let .selectFolder(folder):
            folders.append(folder)
            return []
        // ...
        }
    }
}
```

Step 2: The New View Hierarchy Is Computed from the State

As in the previous example, the changed state causes a new view hierarchy to be computed. In the construction section, we described in more detail how the folder table view controllers are created.

The Elm Architecture Framework

In the implementation details section from before, we saw examples of TEA code that work with entities like messages, the app state, virtual views, etc. The only thing these entities have in common is that they're completely separate from any concepts known to UIKit. Things like UIView instances and UIViewController instances, notifications, and target-action are markedly absent. This is because the TEA framework bridges the gap between all the usual UIKit constructs and the abstractions used in TEA code.

The central component that ties everything together is the Driver class. The driver owns the model, syncs up the virtual view hierarchy with UIKit's view hierarchy, receives view actions, and executes side effects like persisting data in the store. To understand the implementation of the framework, we'll start by examining how virtual views are rendered into UIView and UIViewController instances.

Virtual Views

In our implementation, virtual views are defined as an enum. Each kind of view is a case in this enum:

```
indirect enum View<Message> {
    case _label(Label)
    case _stackView(StackView<Message>)
    // ...
}
```

The data needed to configure each virtual view is stored as an associated value. Since most virtual views have a lot of options, we package them up into structs. For example, the StackView struct looks like this:

```
struct StackView<Message> {
    let views: [View<Message>]
    let axis: UILayoutConstraintAxis
    let distribution: UIStackViewDistribution
    let backgroundColor: UIColor

    init(views: [View<Message>],
        axis: UILayoutConstraintAxis = .vertical,
        distribution: UIStackViewDistribution = .equalCentering,
        backgroundColor: UIColor = .white)
    {
        self.views = views
        self.axis = axis
        self.distribution = distribution
        self.backgroundColor = backgroundColor
    }
    // ...
}
```

The virtual views are transformed into UIView instances by a renderer:

```
struct Renderer<Message> {
    // ...
    mutating func render(view: View<Message>) -> UIView {
        switch view {
        case let ._label(label):
            let l = UILabel()
            render(label, into: l)
            return l
        case let ._stackView(stackView):
            let views = stackView.views.map { render(view: $0) }
            let result = UIStackView(arrangedSubviews: views)
```

```
         render(stackView, into: result)
         return result
         // ...
      }
   }
   // ...
}
```

For a label, the renderer creates a UILabel instance and configures it with the payload data of the ._label case. For a stack view, we first recursively render all its children and then configure the stack view with its arranged subviews and any other options like spacing and layout direction.

To avoid replacing the entire view and view controller hierarchy with each change, the framework makes an effort to reuse existing views on subsequent updates. For example, the update code for a label looks like this:

```
// Renderer
mutating func update(view: View<Message>, into existing: UIView) -> UIView {
   switch view {
   case let ._label(label):
      guard let l = existing as? UILabel else {
         return render(view: view)
      }
      render(label, into: l)
      return l
      // ...
   }
}
```

Only if no UILabel instance exists do we create a new one. Otherwise, we reuse the existing label and just update its properties in the render(_:into:) method:

```
// Renderer
private func render(_ label: Label, into l: UILabel) {
   l.setContentHuggingPriority(UILayoutPriority.defaultHigh, for: .horizontal)
   l.text = label.text
   l.font = label.font
   l.textAlignment = .center
}
```

Dispatching View Actions

To see how the TEA framework delivers messages to the app state's update method in response to view actions, we start by examining how views are created and follow the event flow from there. For example, the play button in the Recordings project is defined as follows:

```
return View<Message>.stackView(views: [
   .stackView(views: [
      // ...
      .button(text: playing ? .pause : .play, onTap: .togglePlay),
   ]),
   // ...
])
```

When we create the virtual button, the second parameter specifies which message should be sent in response to a tap. The renderer uses this information when it creates the actual UIButton instance for the virtual button:

```
// Renderer
private mutating func render(_ button: Button<Message>, into b: UIButton) {
   b.removeTarget(nil, action: nil, for: .touchUpInside)
   if let action = button.onTap {
      let cb = self.callback
      let target = TargetAction { cb(action) }
      strongReferences.append(target)
      b.addTarget(target, action: #selector(TargetAction.performAction(sender:)),
         for: .touchUpInside)
   }
   // ...
}
```

Using UIKit's standard target-action mechanism, we install an internal TargetAction object as the receiver for tap events from the UIButton. The purpose of the TargetAction class is to translate target-action events into messages and forward them to a callback function. All views use the same callback function for all events. This function is passed into the render methods by the driver — for example, during initialization:

```
final class Driver<Model, Message> where Model: Codable {
  init(_ initial: Model,
    update: @escaping (inout Model, Message) -> [Command<Message>],
    view: @escaping (Model) -> ViewController<Message>,
    subscriptions: @escaping (Model) -> [Subscription<Message>],
    initialCommands: [Command<Message>] = [])
  {
    // ...
    strongReferences = view(model)
      .render(callback: self.asyncSend, change: &viewController)
    // ...
  }
  // ...
}
```

The asyncSend method, used as the callback for the renderer, switches to the main queue and invokes the run method:

```
// Driver
func asyncSend(action: Message) {
  DispatchQueue.main.async { [unowned self] in
    self.run(action: action)
  }
}
func run(action: Message) {
  assert(Thread.current.isMainThread)
  let commands = updateState(&model, action)
  // ...
}
```

This is the place where the driver passes the message to the app state. The updateState function the driver calls here is the update function we passed to the driver at initialization.

All view actions are handled like this. Other UIControl instances work in the same way as the button example shown above. Likewise, delegate-based views such as table views are configured with an internal delegate object that converts table view actions into messages sent to the callback function passed in by the driver.

Handling Commands

The final aspect of the TEA framework we'll look at is how it executes the commands returned from the update method. In our current implementation, we have app-specific commands (such as deleting an item from the store) and app-independent commands (such as making a network request).

As an example, we'll walk through the case of deleting an item from the table view in which the app state's update method returns a .delete command:

```
// AppState
mutating func update(_ msg: Message) -> [Command<Message>] {
   switch msg {
   case .delete(let item):
      return [Command.delete(item)]
   // ...
   }
}
```

The driver picks up the commands returned from the update method:

```
// Driver
func run(action: Message) {
   assert(Thread.current.isMainThread)
   let commands = updateState(&model, action)
   refresh()
   for command in commands {
      interpret(command: command)
   }
}
func interpret(command: Command<Message>) {
   command.run(Context(viewController: viewController, send: self.asyncSend))
}
```

The context that's passed into the command's run method provides access to the app's root view controller (e.g. to present a modal alert) and a callback, which can be used to send messages back to the app state once the command has executed.

The implementation of the .delete command simply calls delete on the store:

```
extension Command {
  // ...
  static func delete(_ item: Item) -> Command {
    return Command { _ in
      Store.shared.delete(item)
    }
  }
}
```

The .delete command doesn't make use of the context it gets passed in, but some other commands do. For example, a command that executes a network request would send back a message with the received data once the request has completed:

```
extension Command {
  // ...
  static func request(_ request: URLRequest,
    available: @escaping (Data?) -> Message) -> Command
  {
    return Command { context in
      URLSession.shared.dataTask(with: request) {
        (data: Data?, response: URLResponse?, error) in
        context.send(available(data))
      }.resume()
    }
  }
}
```

As we've seen, there's no magic to the TEA framework. The driver is a simple class, and the implementation of commands is straightforward as well. The most work resides in creating the virtual views and writing the render and update methods for all of them. These can become quite complex, especially when animated updates are involved, as is the case for the table view.

Testing

The key difficulty in application testing is often to find a clean, contained interface that encloses the logic we want to test. Finding these clean interfaces in TEA is surprisingly simple — they're all shown when we construct the driver in the application delegate:

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
  // ...
  let driver = Driver<AppState, AppState.Message>(
    AppState(rootFolder: Store.shared.rootFolder),
    update: { state, message in state.update(message) },
    view: { state in state.viewController },
    subscriptions: { state in state.subscriptions })
  // ...
}
```

The AppState offers the three critical interfaces — used in the closures in this code sample — that we need for testing:

1. Rendering virtual views (provided by the top-level viewController computed property)
2. Performing changes (handled exclusively through the update method)
3. Observing external effects (described by the subscriptions computed property)

Tests in TEA use just one of these functions combined with the AppState to test a narrow slice of application functionality. For example, given a root folder with two items, we can test that a table view controller is built and that the table view contains two items as well. The test starts with the viewController property on AppState and walks through the virtual view hierarchy, down to the relevant tier. Finally, it asserts that the properties have the expected values. Note that we also test that the correct messages will be sent upon cell selection:

```
func testFolderListing() {
    // Construct the AppState
    let vc = AppState(rootFolder: constructTestFolder()).viewController

    // Traverse and check hierarchy
    guard case .splitViewController(let svc, _) = vc else { XCTFail(); return }
    let navController = svc.left(nil)
    let navItem = navController.viewControllers[0]
    XCTAssertEqual(navItem.title, "Recordings")
    guard case .tableViewController(let view) = navItem.viewController
      else { XCTFail(); return }

    // Check structure
    XCTAssertEqual(view.items.count, 2)

    XCTAssertEqual(view.items[0].text, "📁 Child 1")
    XCTAssertEqual(view.items[0].onSelect, .selectFolder(folder1))

    XCTAssertEqual(view.items[1].text, "🔊 Recording 1")
    XCTAssertEqual(view.items[1].onSelect, .selectRecording(recording1))
  }
```

The structure of the test above is similar to the testTableData() test in the MAVB implementation. While view binders and virtual views have very different implementations, they are both abstractions over the real views, and the similarity of the tests reflects that.

To test folder selection, we don't need to render the virtual view hierarchy. After all, in the previous test, we already verified that the .selectFolder message is correctly set for the folder cell. Instead, we start with a known state, call the update method with the .selectFolder message, and verify that the state has changed accordingly:

```
func testFolderSelection() {
  // Construct the AppState
  var appState = AppState(rootFolder: constructTestFolder())

  // Test initial conditions
  XCTAssertEqual(appState.folders.map { $0.uuid }, [rootUuid])

  // Push a new folder
  let commands = appState.update(
    .selectFolder(Folder(name: "Child 1", uuid: uuid1, items: [])))

  // Test results
```

```
    XCTAssert(commands.isEmpty)
    XCTAssertEqual(appState.folders.map { $0.uuid }, [rootUuid, uuid1])
}
```

Navigation tests in MVC and MVVM are notoriously difficult since there is no simple interface in these architectures that encapsulates the navigation state. In the code above, we split the test into two parts: in one test, we verified that the correct view hierarchy is rendered for a given state, and in a separate test, we verified that the state changes correctly for a given message. We don't need to test that the view hierarchy has changed, as this is taken care of by the framework. We also don't need to test that components are connected correctly; this is also done by the framework.

The test above shows a structure similar to the testSelectedFolder() in the MVC+VS implementation. In MVC+VS, however, we had to render the view hierarchy in order to change the state because we wanted to verify that the view hierarchy and state are connected. This is not necessary in TEA.

Unlike in other architectural patterns, it's very easy to test subscriptions in TEA. These tests work similar to those for the view hierarchy. Below, we verify two things: first, we ensure that there is a subscription to the store, and second, that the message is a .reloadFolder message. Note that message in the code below is a function that takes a folder and returns a message:

```
func testFolderStoreSubscription() {
    let appState = AppState(rootFolder: constructTestFolder())
    // Test for a store subscription
    guard case let .storeChanged(message)? = appState.subscriptions.first
        else { XCTFail(); return }

    // Test that the message is a `.reloadFolder`
    let updatedFolder = Folder(name: "TestFolderName", uuid: uuid5)
    XCTAssertEqual(message(updatedFolder), .storeChanged(updatedFolder))
}
```

Just as with testing the view hierarchy, testing subscriptions always follows the same pattern: given a state, call the subscriptions method and verify that the result value is correct. We don't need to test that we're actually subscribed to the Store; the fact that .storeChanged works correctly is tested at the framework level. We also don't need to verify that the message will get delivered, as the driver takes care of this.

In the three kinds of tests we've written thus far (view tests, update tests, and subscription tests), we did not have to invoke a single framework method or class. This is typical: in TEA, we write pure functions and test the output values.

Testing Side Effects

In the test for update, we verified that the returned commands array is empty. This array contains the side effects to be executed. In our implementation of the framework, we modeled Command as a function so that we can provide built-in commands but be open for extension as well.

If we want to test that the correct command is returned, we have two different options. We can execute the command, but that usually takes a lot of setup or is expensive. Alternatively, we can change the type of our update method to return a protocol. For example, consider the update method for our RecordState:

```
extension RecordState {
  // ...
  mutating func update(_ message: Message) -> [Command<Message>] {
    switch message {
    // ...
    case let .save(name: name):
      guard let name = name, !name.isEmpty else {
        // show that we can't save...
        return []
      }
      return [Command.saveRecording(name: name, folder: folder,
        url: recorder.url)]
    // ...
    }
  }
}
```

In this method, we use three different commands. To make the commands testable, we start by grouping them in a protocol (we're only showing a single command for the sake of brevity):

```
protocol CommandProtocol {
  associatedtype Message
  // ...
  static func saveRecording(name: String, folder: Folder, url: URL) -> Self
}
```

Because the definitions of the methods match our Command struct exactly, we can make Command conform without doing any extra work:

```
extension Command: CommandProtocol { }
```

Finally, we can change the type of our update method to return an array of Cs, where C conforms to the new protocol. When we use update to configure our driver, the compiler will infer that C should be equal to Command:

```
extension RecordState {
  // ...
  mutating func update<C>(_ message: Message) -> [C]
    where C: CommandProtocol, C.Message == Message {
    switch message {
    // ...
    case let .save(name: name):
      guard let name = name, !name.isEmpty else {
        // show that we can't save...
        return []
      }
      return [C.saveRecording(name: name, folder: folder, url: recorder.url)]
    // ...
    }
  }
}
```

For testing, we can create an enum that also conforms to CommandProtocol. The definition is mechanical: for each static method in the protocol, we define a single case. For the sake of brevity, we left out the protocol method implementations, which construct and return the cases below:

```
enum CommandEnum<Message>: CommandProtocol {
  // ...
  case _saveRecording(name: String, folder: Folder, url: URL)
  // ...
}
```

In our test, we now enforce that the result type is a CommandEnum, and this is all we need to instruct the compiler to return the correct type. Below, we combine two tests into one: first, we check that when we call save without a name value (because the user didn't enter any text in the modal dialog), we don't get any commands returned. Then we check that if we do provide a value, we get a command to save the recording into the store and persist it to disk:

```
func testCommands() {
  var state = RecordState(folder: folder1, recorder: sampleRecorder)

  let commands: [CommandEnum<RecordState.Message>] =
    state.update(.save(name: nil))
  XCTAssert(commands.isEmpty)

  let commands1: [CommandEnum<RecordState.Message>] =
    state.update(.save(name: "Hello"))
  guard case ._saveRecording(name: "Hello",
    folder: folder1, url: sampleRecorder.url)? = commands1.first
    else { XCTFail(); return }
}
```

These kinds of tests become even more useful when dealing with asynchronous commands — for example, with the request command from the previous section. We can verify that the correct URL is loaded, and that the response is turned into the correct message. In a separate test, we can verify that for the given message, the state changes correctly.

Discussion

The primary benefit of TEA is the absence of a mutable view hierarchy: we describe how the view hierarchy should look for a given app state, but we don't have to write any code to transition from view hierarchy A to view hierarchy B in response to a particular action. Defining our views in such a declarative way eliminates an entire class of potential bugs. We're all familiar with views getting into invalid states — erroneously disabled controls, popovers that aren't dismissed, or loading indicators that don't disappear, to name just a few examples. TEA's consistent and easy-to-follow mechanism for view state updates across all parts of an application alleviates the burden of managing all this state.

The MVC+VS implementation tries to achieve a similar goal by representing all view state as part of the model layer and strictly using the observer pattern for view state updates. However, all this is enforced purely by convention: we could mutate the view hierarchy directly if we wanted to (or if we didn't know about the pattern used in the code base). TEA takes this to the next level: as users of the TEA framework, we have no access to the view hierarchy. The only thing we can do in response to a view or model action is to describe what the view hierarchy should look like. Unlike MAVB, TEA doesn't depend on a reactive programming library.

Similar to how TEA enforces a consistent flow for view updates, it also enforces a consistent pattern for other side effects. For example, within the app state's update method, we can't execute a network request directly and update the view hierarchy when it returns. The only way to achieve these is to return a command from the update method: instead of executing the request ourselves, we provide the driver with a description of what should happen, and the driver then takes care of executing the command and sending a message when it's finished.

However, TEA is not without its drawbacks. One of the problems with implementing a TEA framework on iOS is that UIKit has not been designed to be used in this way. Several UIKit components autonomously change their internal state without us being able to intercept that state change (we're only being notified after the fact). We described this problem already in the MVC+VS chapter, and it shows up for TEA as well. Therefore, the abstraction of UIKit by the TEA framework will always be imperfect.

Another major challenge is how to integrate animations into TEA. When describing the view hierarchy for the current app state, we don't have any information about how we got into this state. However, that's often crucial information for animations (e.g. an animation should run in response to a user interaction, but it shouldn't run if the current state came about programmatically, let's say from state restoration). Furthermore, it's not immediately obvious how intermediate view states (while animating from view hierarchy A to view hierarchy B) should be represented in TEA's app-state driven model. That's an unsolved problem at the time of writing, and it would need further research for us to reach a good solution.

Lessons to Be Learned

One lesson from TEA is that it can be a good idea to model application state explicitly and to update the entire hierarchy when this state changes. TEA takes it to the extreme by applying this idea to the entire application, but the same principle can be applied to certain screens or even to parts of a screen. In the MVC+VS chapter, we talked about a lightweight implementation of view state, which is essentially the same idea.

Declarative Views

Another lesson is the benefit of setting up our views declaratively. Again, TEA goes all the way by introducing a virtual view layer for all kinds of views. However, we can use the same idea and apply it only to those components for which it's easy to implement and which are used regularly.

In Swift Talk #07, we demonstrated a technique using enums to define stack views declaratively. We can start by defining the kinds of elements we want to put into the stack view, i.e. its arranged subviews:

```
enum ContentElement {
  case label(String)
  case button(String, () -> ())
}
```

We then implement a view property in an extension on ContentElement to create an actual UIView out of these enum cases:

```
extension ContentElement {
  var view: UIView {
    switch self {
    case let .label(text):
      let label = UILabel()
      label.text = text
      return label
    case let .button(title, callback):
      return CallbackButton(title: title, onTap: callback)
    }
  }
}
```

The CallbackButton class is a wrapper around UIButton that accepts a callback. Please see the transcript of Swift Talk #07 for the implementation details.

We then use the view property on the content elements to create a convenience initializer on UIStackView:

```
extension UIStackView {
  convenience init(vertical: [ContentElement]) {
    self.init()
    translatesAutoresizingMaskIntoConstraints = false
    axis = .vertical
    spacing = 10

    for element in elements {
      addArrangedSubview(element.view)
    }
  }
}
```

This allows us to quickly set up stack views like so:

```
let stackView = UIStackView(vertical: [
  .label("Name: \(recording.name)"),
  .label("Length: \(timeString(recording.duration))"),
  .button("Play", { self.audioPlayer.play() }),
])
```

This currently only works for static content. If we wanted to render different content elements, we'd have to create a new stack view and swap it in for the old one. For many use cases, this is not a problem at all. But we could also create a method on UIStackView that updates the existing arranged subviews with new data from an array of content elements if the type of all subviews match. Otherwise, it could replace the existing subviews with new ones.

In combination with the lightweight view state approach outlined in the MVC+VS chapter, we can approximate TEA's use of declarative virtual views for small parts of an existing app. We have shown a more complex example of using these techniques in the Building a Form Library series on Swift Talk.

Describing Side Effects

TEA has an interesting way of dealing with side effects like executing a network request. Due to how the architecture is set up, there is no way for us to directly perform such a task from the app state's update method. Because of this, TEA introduces the concept of commands: to execute a network request, we return a command describing this request and leave the actual execution to the driver.

Although we don't face these kinds of restrictions in a standard MVC code base, the idea of TEA's commands is applicable as well: instead of entangling the information about what should be done with the code that actually performs the task, we strictly separate the two. This is especially useful for asynchronous tasks: it disentangles the simple, synchronous parts from the more-difficult-to-test asynchronous code. An example of this pattern is the technique we used for a tiny networking library in the very first Swift Talk episode.

The idea is simple: we create a struct that contains all the information needed to make a network request to a certain endpoint — including the function that can turn the response data into a useful data type:

```
struct Resource<A> {
   let url: URL
   let parse: (Data) -> A?
}
```

This struct could contain much more information, such as the request type and the content type. Abstracting network requests like this makes it simple to test our parsing code: we just have to test if the resource's parse function returns what we expect for a given input, and the test for that is synchronous. The asynchronous task of actually making the request can now be written once:

```
final class Webservice {
   func load<A>(_ resource: Resource<A>, completion: (A?) -> ()) {
      URLSession.shared.dataTask(with: resource.url) { data, _, _ in
         let result = data.flatMap(resource.parse)
```

```
            completion(result)
        }.resume()
    }
}
```

Although this example lacks many features (good errors, authentication, etc.), the principle is clear: everything that varies from request to request is pulled into the resource struct, and the web service's load method is left with the sole task of interacting with the networking APIs.