Functional Reactive Programming on iOS

Ash Furrow

Functional Reactive Programming on iOS

Functional reactive programming introduction using ReactiveCocoa

Ash Furrow

This book is for sale at http://leanpub.com/iosfrp

This version was published on 2014-01-05

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

Also By Ash Furrow

Your First iOS 7 App

Contents

Acknowledgements

Thanks to Justin Spahr-Summers and Josh Abernathy, Dave Lee, as well as the entire ReactiveCocoa team for putting together such a great framework and community of developers. Their work has made me, and I'm sure countless others, better developers.

Philosophy

This is going to be a lofty chapter. "Why?", you ask. "I thought this was a book about programming? I want my money back!" Hold on there! This is a book for developers who want to get better at programming, and we need to talk about *why* we want to get better first.

Why do we want to improve, especially given programmers are known to be lazy? The reason we want to improve our skills is so that we can be *more lazy*. We want to write less code, or write code faster. Functional reactive programming can help us accomplish these goals, but it means seriously stepping outside of our comfort zones.

All programs are meant to accomplish some tasks. Most developers are trained in what's called *imperative programming*. This paradigm relies on programmers to think about *how* they want their programs to accomplish these tasks: developers write many instructions that modify the program's state. If the programmer writes the correct instructions in the correct order, then the program's output will correctly accomplish its task.

Sounds mundane.

Why are we all trained to think in terms of *how*? I imagine we all program imperatively because, under the hood, we know that that's how computers actually work. The program counter of our CPU dutifully marches forward: fetch, execute, fetch, execute. How boring.

In contrast, *declarative programming* frees programmers from the minutiae of *how* to accomplish some tasks and lets the focus solely on *what* it is they want to accomplish. Declarative programming is an umbrella term for several other paradigms, which we'll discuss shortly.

From Wikipedia:

> Declarative programming is a non-imperative style of programming in which programs describe their desired results without explicitly listing commands or steps that must be performed. Functional and logical programming languages are characterized by a declarative programming style.

Functional reactive programming falls under the Declarative Programming paradigm; it is the main focus of this book.

Functional Reactive Programming

Functional Reactive Programming is a paradigm that is a combination of two declarative sub-paradigms: *functional* and *reactive*. Let's deal with reactive programming first, since it's much simpler.

Reactive programming is best described in terms of a spreadsheet. Suppose we have a cell, `A`, which is the result of two other cells' sums (cells `B` and `C`). When the values contained by either `B` or `C` change, then those changes are propagated through the spreadsheet and the value of `A` changes. In short, we *declare* that `A` is always the sum of `B` and `C`. Always always always. We don't care how it happens – just that we can rely on its truthfulness.

Next we need to define functional programming. It's difficult to do, as anyone who has tried to Google the term will tell you. The functional paradigm is a framework with which to we can structure our programs. At its core, functional programming is about treating functions as first-class objects within your language.

There is no mutable state in functional programming. A function, `f`, always yields the same output given the same input. Always.

Scary, I know, since we programmers rely on the mutability of state to write our applications. It's scary to think about a world where, after assigning a value to a variable, you can't re-assign it. Functional programming seems excessive, and in many ways, it is. There are parts of programming (user input, network I/O, etc) that are just not easily structured using a functional paradigm. That's where the reactive component of functional reactive programming comes in.

Functional reactive programming is the sweet-spot – a compromise between imperative programming and functional programming. It lets us have our cake and eat it, too, like the hungry little malnourished programmers we are.

Functional reactive programming treats user input as functions whose results change over time. In this light, consider the function `f` that we talked about earlier. `f` is always supposed to return the same value when sent the same parameters. If the parameter is `time`, then `f` never has to return the same value, since time is always changing. It's kind of a cheat, but remember, we're building up a framework within which we can think. We're allowed to cheat.

Conclusion

The key take aways in this chapter are:

- We're here to learn functional reactive programming so we can be more efficient.
- Declarative programming frees us from having to think about *how* to accomplish tasks and lets us focus only on the tasks themselves.
- Functional reactive programming is the marriage of functional and reactive programming.

I've never been a huge fan of teaching theory in a vacuum. We're all developers with stuff to get done, so I've tried to take up as little of your time discussing philosophy as possible. We'll dig into some code in the next chapter.

Functional Programming with `RXCollections`

So this is a book about functional reactive programming, right? Well, just like we need to learn to walk before we can run, we need to learn how to program functionally before we can use functional reactive programming effectively.

Higher-Order Functions

One of the key concepts of functional programming is that of a "higher-order function". According to Wikipedia, a higher-order function is a function that satisfies these two conditions:

- It takes one or more functions as input.
- It outputs a function.

In Objective-C, we often use blocks as functions.

We don't have to look very far to find some higher-order functions baked into Foundation for us by Apple. Consider a simple array of numbers:

```
1  NSArray *array = @[@(1), @(2), @(3)];
```

We might want to enumerate over the contents of that array, doing something with each element. "Fine", you say, "I'll just write a `for` loop."

Stop right there, buddy. As I've written before, stop writing `for` loops. There is a higher-order function of `NSArray`'s that we can use, instead.

This code:

```
1  for (NSNumber *number in array) {
2      NSLog(@"%@", number);
3  }
```

... is equivalent to writing the following, using a higher-order function.

```
[array enumerateObjectsUsingBlock:^(NSNumber *number, NSUInteger idx, BOOL *stop)\
 {
    NSLog(@"%@", number);
}];
```

"But why?", you ask. "Isn't it more code?" Well, it is, but this is the first step in our path to functional enlightenment. See how, like in the last chapter, we've abstracted *how* to accomplish the task and focused only on the task itself? There is a pay-off coming, trust me.

In practice, higher-order functions are abstractions for things we do all the time. Unfortunately for us, that's about the extent of the higher-order functions in Foundation. For more, we have to turn to the open source community.

Installing `RXCollections`

My friend Rob Rix has written an excellent library of higher-order functions in Objective-C called `RXCollections`.

First we'll need an Xcode project in which to play around. Create a new one called "Playground". Pick "Single View Application" as the template for your new project. We'll be putting most of the code we play with in our app delegate, anyway. In this book, I'll be using "FRP" as my class prefix.

Next we'll need to install `RXCollections`. I'm going to take the CocoaPods route since it's the easiest. Make sure you have CocoaPods installed on your machine by running the following command:

```
sudo gem install cocoapods
```

Enter your password when prompted. Once CocoaPods has been installed, use `cd` to navigate to the directory where your newly created Xcode project lives. Type the following command:

```
pod init
```

This will generate an empty `Podfile` in the directory for you.

```
# Uncomment this line to define a global platform for your project
# platform :ios, "6.0"

target "Playground" do

end

target "PlaygroundTests" do

end
```

Open your favourite text editor, which is undoubtedly vim, and uncomment the line '# platform :ios, "6.0" and add the line pod 'RXCollections', '1.0'' to the "Playground" target.

```
platform :ios, "6.0"

target "Playground" do

pod 'RXCollections', '1.0'

end

target "PlaygroundTests" do

end
```

Great. Return to the command line and run the following command.

```
pod install
```

This will install RXCollections and create a new Xcode workspace file for you. Close the Xcode project and open the Xcode workspace, instead.

Open your application delegate's .m file and add the following `#import` to the top of the file.

```
#import <RXCollections/RXCollection.h>
```

In your `application:didFinishLaunchingWithOptions:` method, create the array we talked about earlier.

```
NSArray *array = @[@(1), @(2), @(3)];
```

We're now ready to get our hands dirty.

Map

This first higher-order function we're going to play with is "map". At a high-level, map takes a list and turns it into another list of the same length, "mapping" each value in the original list into a new value in the resulting list. A map to square numbers might result in the following:

```
map (1, 2, 3) => (1, 4, 9)
```

Of course, that's just pseudocode, and a higher-order function returns another function, not a list. So how do we work with map in RXCollections?

We do so with the `rx_mapWithBlock:` method.

```
NSArray *mappedArray = [array rx_mapWithBlock:^id(id each) {
    return @(pow([each integerValue], 2));
}];
```

This accomplishes the same mapping. If we log the array, we'll see the following contents:

```
(
    1,
    4,
    9
)
```

Perfect! Note that `rx_mapWithBlock:` isn't a *true* map, since it's not technically a higher-order function (it doesn't return a function). Subsequent commits to the library have addressed this, and we'll see how maps work in the context of ReactiveCocoa in the next chapter.

Notice that `rx_mapWithBlock:` returns a *new* array, not modifying the values in the original. In this sense, Foundation classes mesh really well with a functional paradigm, since their classes are immutable by default.

Imagine all the code we'd have to write in order to accomplish this imperatively.

```
NSMutableArray *mutableArray = [NSMutableArray arrayWithCapacity:array.count];

for (NSNumber *number in array) {
    [mutableArray addObject:@(pow([number integerValue], 2))];
}

NSArray *mappedArray = [NSArray arrayWithArray:mutableArray];
```

That's a lot more code, not to mention the useless `mutableArray` local variable now polluting our lexical scope. Gross.

So you can see that map is a useful function when you have a list of things and you want to turn it into another list of things.

Filter

Another one of the key higher-order methods we're going to be using when it comes to ReactiveCocoa is the filter method. Filtering a list just returns a new list containing all of the original entries, minus the entries that didn't return true from a test. Let's take a look at this in practice.

```
NSArray *filteredArray = [array rx_filterWithBlock:^BOOL(id each) {
    return ([each integerValue] % 2 == 0);
}];
```

`filteredArray` is now just the array `@[@(2)]`. In contrast, this is the work we would have to do without this abstraction.

```
NSMutableArray *mutableArray = [NSMutableArray arrayWithCapacity:array.count];

for (NSNumber *number in array) {
    if ([number integerValue] % 2 == 0) {
        [mutableArray addObject:number];
    }
}

NSArray *filteredArray = [NSArray arrayWithArray:mutableArray];
```

Starting to get the picture, right? You've probably written code like this, and it's companion in the previous section, a hundred times or more. How much of our work day-to-day is involved in doing things like mapping or filtering lists? Lots of it! By using higher-order functions like map and filter, we're able to abstract that grunt work away.

Fold

Fold is an interesting higher-order function – it combines each entry in a list down to a single value. For this reason, it's often referred to as "combine".

A simple fold can be used to combine the integer values of each member of our `array` to calculate their sum.

```
NSNumber *sum = [array rx_foldWithBlock:^id(id memo, id each) {
    return @([memo integerValue] + [each integerValue]);
}];
```

This yields the value of `@(6)`. Each member of the array is called in sequence, with the `memo` parameter being the return value of the previous invocation of the block (the initial value is `nil`).

This isn't really that interesting, though. What's more interesting is if we can supply an initial value for the `memo` property. Luckily, there is such a method.

```
[[array rx_mapWithBlock:^id(id each) {
    return [each stringValue];
}] rx_foldInitialValue:@"" block:^id(id memo, id each) {
    return [memo stringByAppendingString:each];
}];
```

The result of this code is `@"123"`. Let's take a look at how. We begin by mapping the `NSNumber` instances into their `NSString` representations. Then we perform a fold, passing in the empty string as the initial `memo` value.

Could this result be computed without `RXCollections`? Of course. But this is an explicit, "what, not how" approach that frees us from having to think about the "how" when we write it, and more importantly, when we read it.

Performance

The previous section, and in particular, the previous code example may have left you wondering about performance. With long arrays, for example, creating an intermediate string representation for each value as it's appended to the previous result may take longer than simply writing it out imperatively.

That may be the case. Luckily, computers (even iPhones) are powerful enough that in most of the cases, this performance hit is negligible. The CPU's time is cheap, but your time is expensive. It's better to sacrifice one for the other, especially when you can always go back and make it more efficient later if it does turn out to be a performance bottleneck.

Conclusion

We've seen over this past chapter how we can operate on lists without needing mutable variables. While `RXCollections` *may* be using mutable variables under the hood, we don't worry about that because it's abstracted away for us. It's unimportant to the tasks at hand: mapping, filtering, and folding.

(That's not to say you shouldn't become familiar with the source code of `RXCollections`, just that you don't need to in order to accomplish your tasks.)

We also saw, in the last example, how we can chain operations together to get a more complex result. We're going to be talking more about chaining operations together in the next chapter – in fact, it's one of the main principles of using ReactiveCocoa.

We're going to be discussing more about map, filter, and fold in the next chapter. Not only will we be using these higher-order functions on lists, but we're also going to see them applied to *streams*, which will be introduced next. We'll also take a look at some other higher-level functions, now that you're familiar with the concepts introduced in this chapter.

Introduction to ReactiveCocoa

The methods we learnt in the last chapter – map, filter, and fold – are functional methods. We'll see them again throughout this chapter. However, this chapter is about ReactiveCocoa and functional reactive programming, which requires a bit more of an introduction.

Installing ReactiveCocoa

ReactiveCocoa can be installed one of two ways: with CocoaPods or as a submodule. ReactiveCocoa doesn't *officially* support CocoaPods, but the open source community has provided the CocoaPods support, which is what we'll be using. If you'd like to use a submodule, instead, feel free to follow the official instructions and check out the `2.0` tag.

To install ReactiveCocoa using CocoaPods, open the podfile we created in the last chapter and remove the `RXCollections` line. Replace it with `pod 'ReactiveCocoa', '2.0'` for both targets. Your podfile should look like the following.

```
platform :ios, "6.0"

target "Playground" do

pod 'ReactiveCocoa', '2.1.4'

end

target "PlaygroundTests" do

pod 'ReactiveCocoa', '2.1.4'

end
```

Note that we're using `2.0`, and not the latest version. Re-run `pod install`, which will remove `RXCollections` from your project and install ReactiveCocoa. Any `#import` statements or use of `RXCollections` methods will break, so remove them from the app delegate file.

In this chapter, we're going to be putting all of our code in the view controller's implementation file, instead of the app delegate, so open that file now. Don't forget to add the following `#import`.

```
#import <ReactiveCocoa/ReactiveCocoa.h>
```

Streams and Sequences

A *stream* is an abstraction for a series of values. You can think of a stream like a pipe, where values are inserted at one end and come out the other. It's impossible to access past values, or even the current value unless you're at the end of the pipe when it comes out. That's OK, as we'll see.

A series of values, eh? Kind of like a list, or in our case, an array. In fact, we can easily turn an `NSArray` into a stream with the `rac_sequence` method.

```
NSArray *array = @[@(1), @(2), @(3)];

RACSequence *stream = [array rac_sequence];
```

Wait. *Sequences*? I thought we were dealing with streams? Well, we are. A sequence is one of two specific types of streams. In fact, `RACSequence` is a subclass of `RACStream`.

At any rate, what do we do with our `stream`? Well, I'd like to show you how we can apply the same methods from the last chapter on this new stream. Let's apply our same squaring map.

```
[stream map:^id(id value) {
    return @(pow([value integerValue], 2));
}];
```

Note: Like `NSArray`, streams cannot contain nil.

That's great, but `map` just returns a stream. How would we get that back to an array? Well, luckily `RACSequence` has this handy method called `array`.

```
NSLog(@"%@", [stream array]);
```

This will print out our mapped array. It's a little more work to do that using straight `RXCollections`, but I want to illustrate how you can work with streams in the same way.

Of course, we can combine the above method invocations so we're not polluting our local variable scope.

```
NSLog(@"%@", [[[array rac_sequence] map:^id(id value) {
    return @(pow([value integerValue], 2));
}] array]);
```

To recap, we've got an array, which is a list of values. Then we're turning that array into a sequence, which is a kind of stream. We're mapping that sequence, which returns a new sequence, then turning it back into an array.

Sequences are, by default, lazily-loaded. They're *pull-driven*, providing values whenever they're asked for them. The `array` method forces evaluation of each of the members of the sequence.

Let's take a look at filtering. In order to filter our array using ReactiveCocoa, we need to again turn it into a sequence to filter it.

```
NSLog(@"%@", [[[array rac_sequence] filter:^BOOL(id value) {
    return [value integerValue] % 2 == 0;
}] array]);
```

Finally, we can see how to to fold values from a sequence down to a single value.

```
NSLog(@"%@", [[[array rac_sequence] map:^id(id value) {
    return [value stringValue];
}] foldLeftWithStart:@"" reduce:^id(id accumulator, id value) {
    return [accumulator stringByAppendingString:value];
}]);
```

In this case, we're chaining two operations on sequences together. This is going to become a key concept when we discuss signals in the next section.

ReactiveCocoa has a concept of a *left* fold and a *right* fold. A left fold will traverse an array from its beginning to its end, where a right fold will traverse it backward. The nomenclature is an allusion to programming languages with list comprehension, which Objective-C does not have.

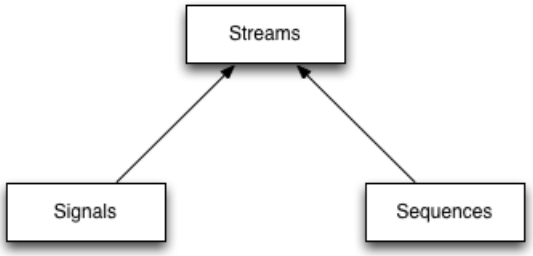
Make sure you understand what's going on here. It's really important before moving on.

Signals

A *signal* is another type of stream. In contrast to sequences, signals are *push-driven*. New values are pushed through the pipe and cannot be pulled. They abstract data that will be delivered in the future.

Signals send three different types of values. *Next* values are exactly what the appear to be: the next value sent down the pipe. *Error* values indicate that a signal could not complete successfully. They are relatively rare, and we'll see in the next chapter how to use them. *Completion* values indicate that a signal completed successfully. Again, we'll see how to use them in the next chapter. It's important to note that once an error or completion value is sent via a signal, *no other values will be sent*. Additionally, only one of either error or completion will be sent – never both.

Signals are one of the core components of ReactiveCocoa. There are signal selectors built-in to UIKit components. For example, `UITextField` has a `rac_textSignal` whose values are sent with every new key press in a text field. We're going to see in the next section how to use signals to perform work.



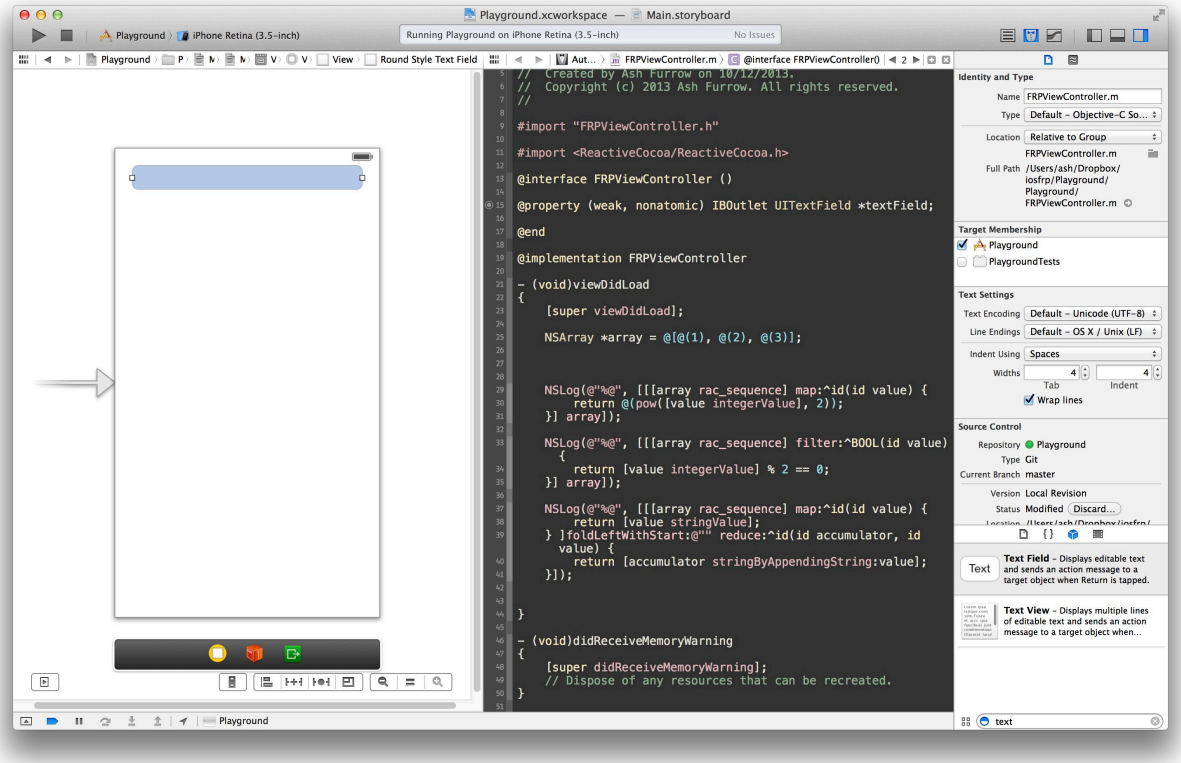
Class diagram

Signals can also be chained and transformed. When we map or filter a stream, we create a new stream. This stream can subsequently be mapped, filtered, and fiddled with all we like. We'll see more about chaining in the next chapter.

Subscriptions

Subscriptions are made on streams – most often signals – when you want to be notified that a new value is sent (either next, error, or completion). Signals are often used to have side effects, such as the following example.

Let's add a text field to our view controller's view and connect it to an `IBOutlet`. I'm going to do this using the built-in Storyboard and Assistant Editor, but you do whatever you like.

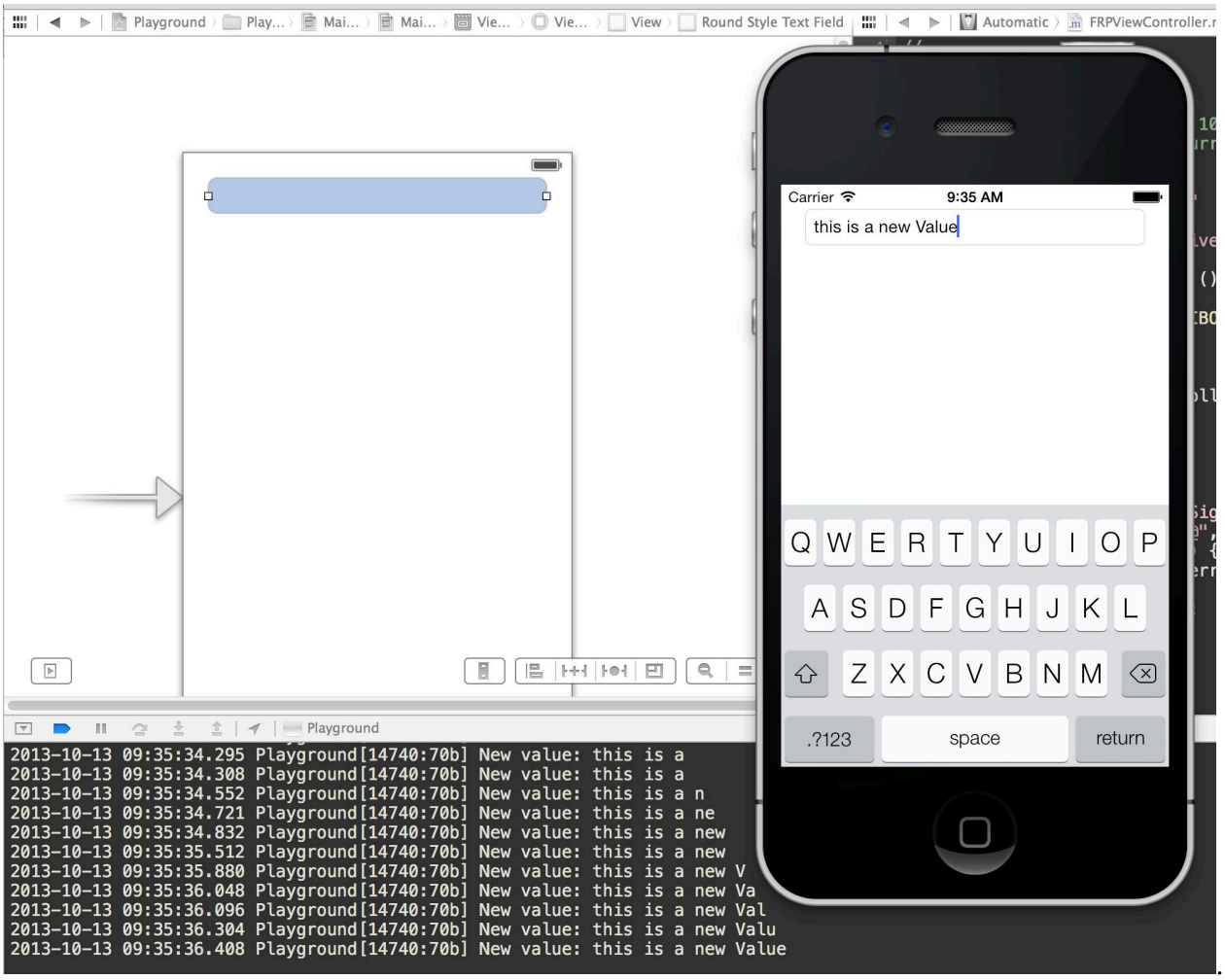


Adding a text field

Add the following lines of code to `viewDidLoad`. They *subscribe* to the `rac_textSignal` of our new text field.

```
[self.textField.rac_textSignal subscribeNext:^(id x) {
    NSLog(@"New value: %@", x);
} error:^(NSError *error) {
    NSLog(@"Error: %@", error);
} completed:^{
    NSLog(@"Completed.");
}];
```

Build and run the application and type something into the text field. Every time a new value is entered into the text field, the next value is sent down the pipe. Then, our subscription block was executed for each new value.



Interestingly, this particular signal doesn't typically send error values and only sends a completion value when it's deallocated, so those subscription blocks will not be called often or at all. We can simplify our code by using a convenience method on `RACSignal`, `subscribeNext:`.

```
[self.textField.rac_textSignal subscribeNext:^(id x) {
    NSLog(@"New value: %@", x);
}];
```

Neato. A lot less code.

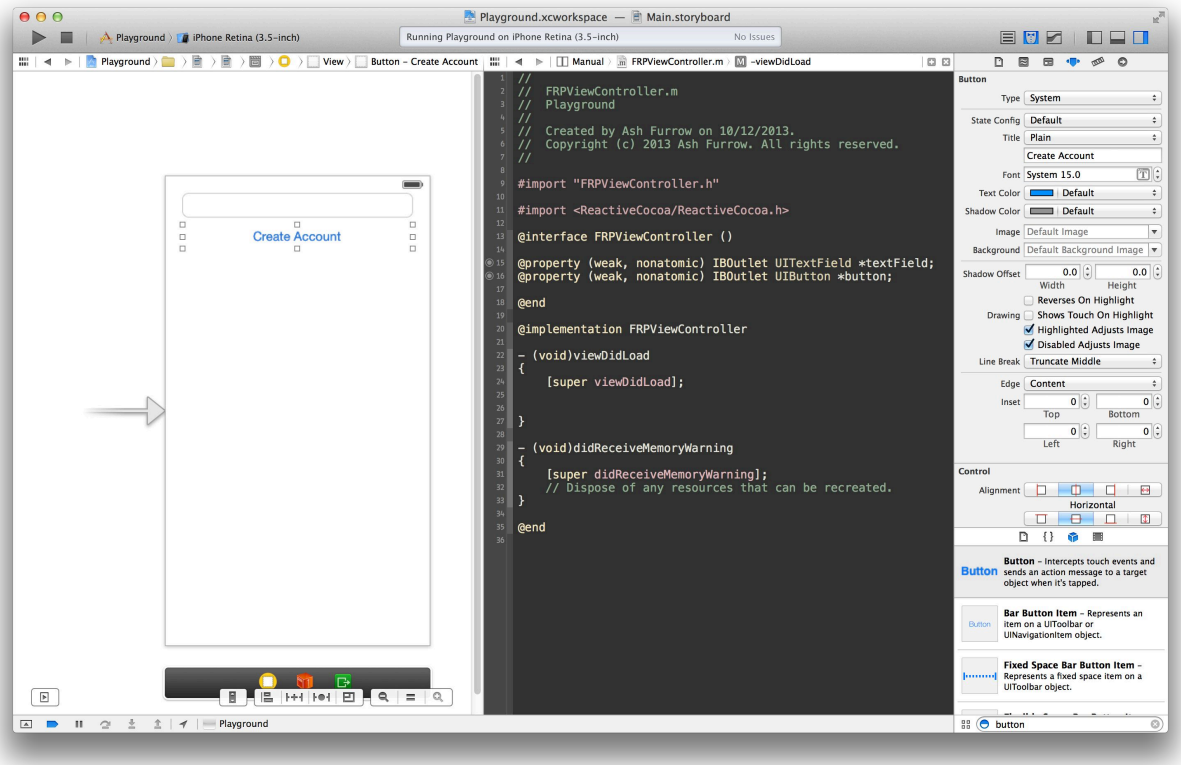
When you subscribe to a signal, you actually create a subscription object. This object is automatically retained for you, and retains the signal its subscribing to, as well. You can manually *dispose* of the subscriber, if you want, but this is not typical behaviour. We'll see in the next chapter how disposing of signals can be helpful when used with reused views (such as collection view or table view cells).

Deriving State

Deriving state is another one of the core components of ReactiveCocoa. Rather than have a property on a class that is set to new values as the state changes, we can abstract that property into a stream. Let's take the previous example and augment it with derived state.

Let's pretend that our view is a form for creating some account and we only want to allow email addresses that contain an '@' symbol. When, and only when, a valid user name has been entered, then a button's enabled state will be `YES`. We also want to provide feedback to the user in way of the text colour of the text field.

Let's first add that button to our view and create an `IBOutlet` for it.



Added a button

Next, we'll want to *bind* our button's `enabled` property to a signal that we'll create.

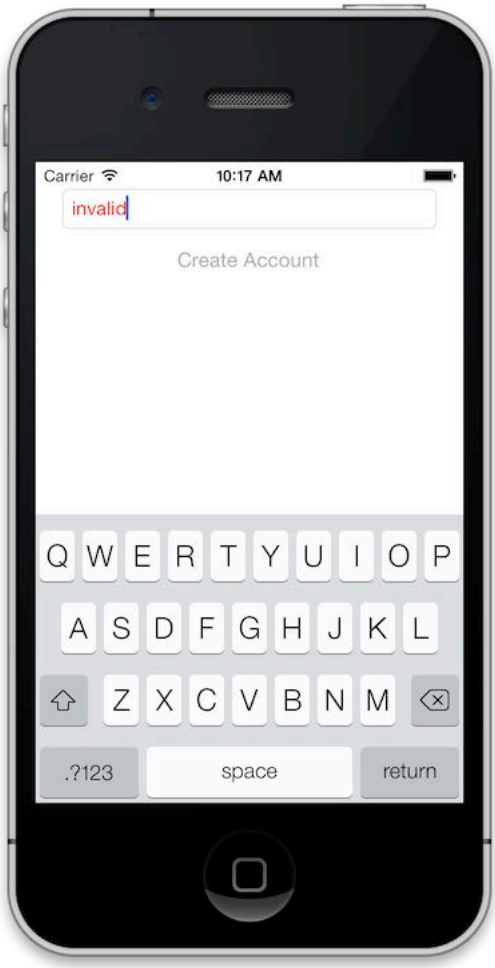
```
1  RAC(self.button, enabled) = [self.textField.rac_textSignal map:^id(NSString *valu\
2  e) {
3      return @([value rangeOfString:@"@"].location != NSNotFound);
4  }];
```

Note that we'll see later how we can use *commands* with buttons to better bind its `enabled` property.

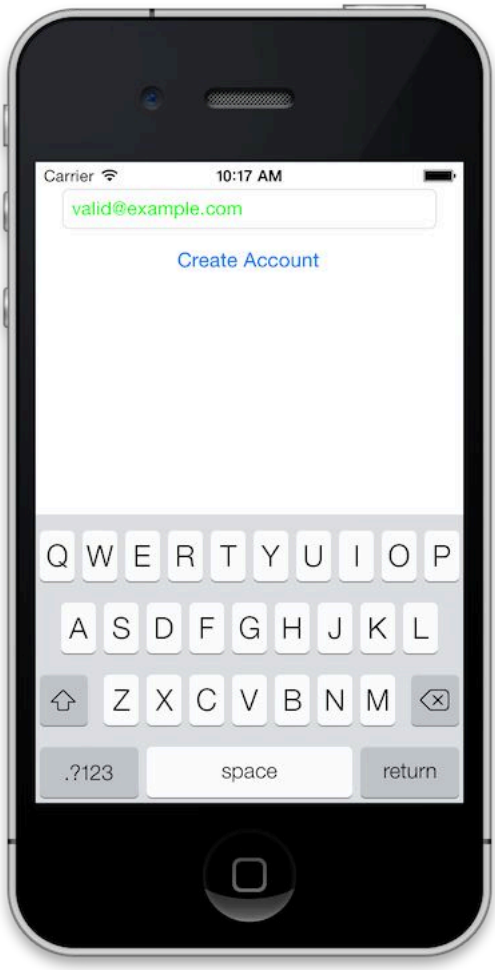
The `RAC()` macro takes two arguments: an object and a key path of that object. It then performs a one-way binding of the right-hand value of the expression to the key path in question. Values must be `NSObjects`, which is why we wrap our boolean expression in an `NSNumber`.

But what about the text colour? We can actually re-use the signal from earlier with a little bit of refactoring.

```
RACSignal *validEmailSignal = [self.textField.rac_textSignal map:^id(NSString *va\
lue) {
    return @([value rangeOfString:@"@"].location != NSNotFound);
}];

RAC(self.button, enabled) = validEmailSignal;
RAC(self.textField, textColor) = [validEmailSignal map:^id(id value) {
    if ([value boolValue]) {
        return [UIColor greenColor];
    }
    else {
        return [UIColor redColor];
    }
}];
```



Invalid email address



Valid email address

Great! See how we're re-using the `validEmailSignal`? That's a very common pattern in ReactiveCocoa. We're also not writing any code outside of our `viewDidLoad` method, which is also very common.

Commands

We mentioned in the last section that binding the `enabled` property of a `UIButton` was not the best idea. That's because `UIButton` has been augmented by a ReactiveCocoa category, adding a *command*. We'll talk about what commands are in this section. The important thing to note is that a button's `rac_command` property takes care of the `enabled` state for us.

Quoting from the ReactiveCocoa documentation:

> A command, represented by the RACCommand class, creates and subscribes to a signal

> in response to some action. This makes it easy to perform side-effecting work as the user interacts with the app.
>
> Usually the action triggering a command is UI-driven, like when a button is clicked. Commands can also be automatically disabled based on a signal, and this disabled state can be represented in a UI by disabling any controls associated with the command.

Commands are useful when you want a signal value sent in response to a user interaction event. The command's signal can be subscribed to later on to process the output of the signal that we return. It's a little confusing, but we're going to see how to use commands in practice in Chapter 5.

For now, replace the binding of the `enabled` property on our button to the following.

```
self.button.rac_command =
    [[RACCommand alloc] initWithEnabled:validEmailSignal signalBlock:^RACSignal *\
(id input) {
    NSLog(@"Button was pressed.");
    return [RACSignal empty];
}];
```

The signal block is executed whenever the button is pressed, and the `rac_command` property takes care of binding the enabled signal to the `enabled` state of the button (in fact, if we had kept our original code, we would have caused an error with two bindings to the same property).

But what about the return value? Well, we need to return a signal that will be sent down the `executionSignals` pipe belonging to the `RACCommand`. This lets you return a *signal* representing some work that will need to be done as the result of the button press. The button will remain disabled until that signal returns its complete value (`empty` returns this value immediately). Because we're simply logging the result of the button press, we're returning an empty signal in this case.

We'll return to `RACCommand` and its uses in chapter 5.

RACSubject

A `RACSubject` is an interesting type of signal. It's the "mutable state" of the ReactiveCocoa world. It's a signal that you can manually send new values through. For this reason, its use is not recommended except in specific cases.

We're going to see in the next chapter how subjects can be used to marry non-reactive code with code written in ReactiveCocoa.

Hot and Cold Signals

Signals are typically *lazy*, meaning that they only do work and send signals when someone has subscribed to them. With each additional subscription, work is re-done. For trivial operations, this is acceptable, and in fact, desirable. In ReactiveCocoa nomenclature, this type of signal is said to be *cold*.

Sometimes, however, we want work to be done immediately. This type of signal is called a *hot* signal. It's very rare to use a hot signal.

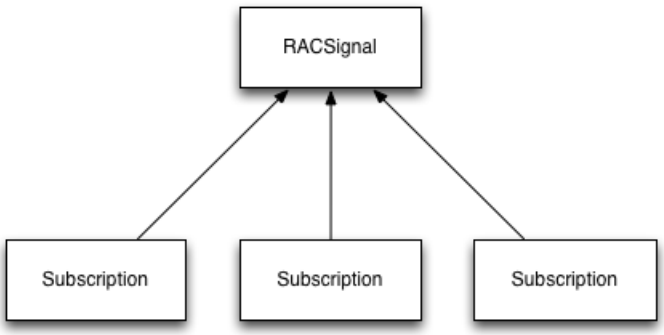
The difference is subtle, yet important. We're going to see in the next chapter how we can use hot signals to our advantage.

Multicasting

Multicasting is a term used to refer to when a single subscription is shared among any number of subscribers. Signals, by default, are cold (as we described in the last section). It's sometimes not desirable to have a cold signal that performs work each time it's subscribed to. This is often done when the side effects or work performed when subscribing is expensive or should only otherwise be done when appropriate. Network requests come to mind.

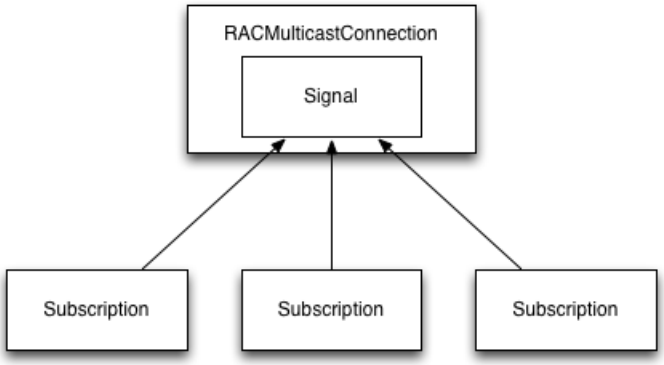
So instead we create a `RACMulticastConnection` from that signal. You can do so either by using the `publish` method on `RACSignal`, or the `multicast:` method. The former method creates a multicast *connection* for you. The latter method does so, as well, but also takes a RACSubject argument. This subject is manually sent the values from the underlying signal whenever it is called. Then, anyone interested in the values sent by the underlying signal subscribe to the connection's signal, instead (if you supplied a subject, the signal happens to be that subject).

To illustrate the difference, consider the following diagram.



Multiple subscriptions

Since the signal is cold by default, each new time a subscriber is added, its work is performed. In cases that that's not desirable, we use the multicast connection.



Multicast connection

The multicast connection subscribes to the signal that it's created with and, when it's passed new values, sends those values through to the signal (which is exposed as a public property). You can subscribe to this signal as many times as you want, and the work performed upon subscription is only done once.

Conclusion

We've covered a lot in this chapter. It's difficult to explain some of these topics, especially the advanced ones, in a vacuum. In the next chapter, we're going to apply the knowledge we gained here and hopefully make it stick. We're going to explore not only the uses of the topics we've seen here, but also acquire some best practices when using ReactiveCocoa.

ReactiveCocoa in Practice

In this chapter, we use ReactiveCocoa for the first time in an actual app, inside of a real-life context. We're going to build a simple 500px iPhone application. 500px is like Flickr, but for only your best shots. I like to use their API for two reasons: the pictures look awesome, and while I worked there, I wrote an iOS SDK for their API, so I'm familiar with it.

We're going to tackle this chapter in three parts. First, we'll come up with a basic implementation of our app, FunctionalReactivePixels. Subsequently, we'll add some new view controllers and do some more data-loading to further demonstrate the basics. Finally, we'll revisit the app to examine opportunities to eliminate more state and to use more functional reactive programming.

This chapter is going to be a lot of fun. It certainly was for me to write. The final results of our FunctionalReactivePixels app is available on GitHub. Unfortunately, some of the intermediate steps aren't in the end result. However, if you follow along with the chapter code, you should be fine.

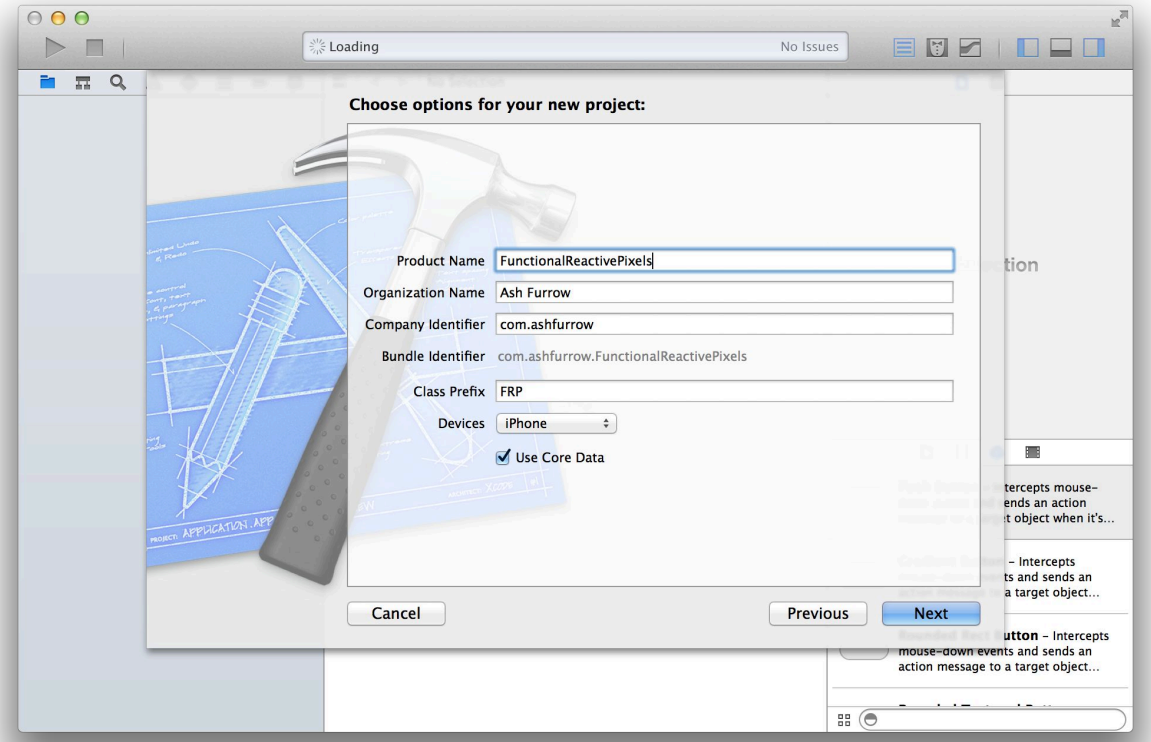
Basics of FunctionalReactivePixels

FunctionalReactivePixels is going to be a simple app for viewing popular photos on 500px. Once we're done this section, the main view of the app will look like the following.

We'll also have a full-sized photo view that looks like this.

This app is going to use collection views. Don't worry if you don't have much experience with them – they're just like table views and our use of them will be very straightforward. If you're interested in learning more, check out one of my other books.

We're going to use CocoaPods again to manage our dependencies, so create a new Xcode project now. I like to start with the "Empty" template to have complete control over the view controller hierarchy.



First, we're going to create a UICollectionViewController subclass called `FRPGalleryViewController`. Create this file with Xcode's New File dialogue. We'll also create a `UICollectionViewFlowLayout` subclass named `FRPGalleryFlowLayout`.

`#import` the new flow layout's header in the view controller's implementation file and then override `FRPGalleryViewController`'s `init` method.

```
- (id)init
{
    FRPGalleryFlowLayout *flowLayout = [[FRPGalleryFlowLayout alloc] init];

    self = [self initWithCollectionViewLayout:flowLayout];
    if (!self) return nil;

    return self;
}
```

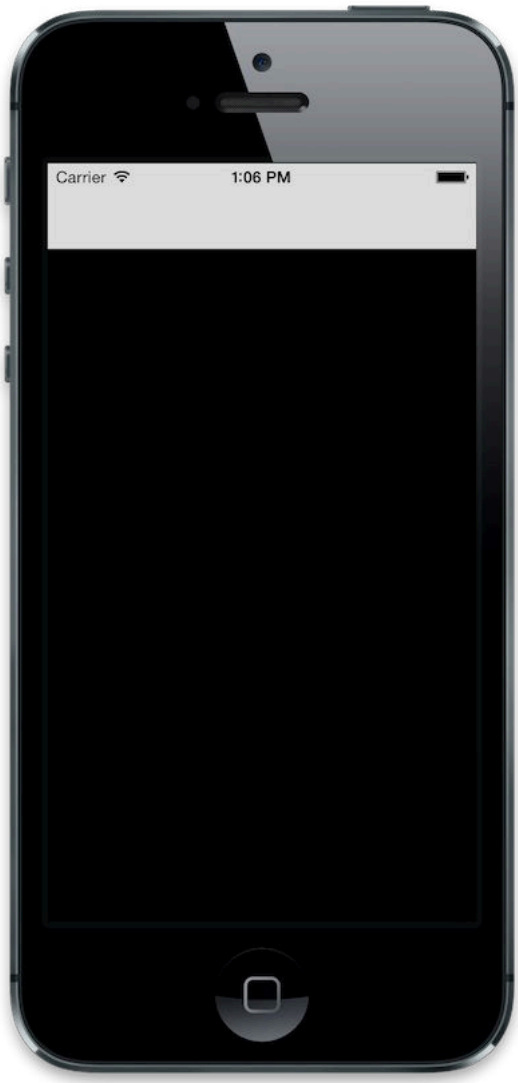
This will initialize ourself with the flow layout as the collection view's layout. The flow layout subclass' implementation is very simple; it just sets some properties on itself.

```
@implementation FRPGalleryFlowLayout

-(instancetype)init {
    if (!(self = [super init])) return nil;

    self.itemSize = CGSizeMake(145, 145);
    self.minimumInteritemSpacing = 10;
    self.minimumLineSpacing = 10;
    self.sectionInset = UIEdgeInsetsMake(10, 10, 10, 10);

    return self;
}

@end
```

Awesome. Next, we need to actually get our view controller on the screen. To do so, we're going to go to our application delegate's `application: didFinishLaunchingWithOptions:` method. We want our collection view controller to be *contained within* a navigation controller.

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    self.window.rootViewController = [[UINavigationController alloc]
        initWithRootViewController:[[FRPGalleryViewController alloc] init]];

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

Great. If we ran the app now, we'd see just an empty view.



Let's populate it with some content! Create a `Podfile`, the instructions for doing which are detailed in the previous chapter, with the following contents.

```
platform :ios, "7.0"

target "FRP" do

pod 'ReactiveCocoa', '2.1.4'
pod 'libextobjc', '0.3'
pod '500px-iOS-api', '1.0.4'
pod 'SVProgressHUD', '0.9'

end
```

```
11
12 target "FRPTests" do
13
14 end
```

We're going to add some tests in the next chapter. For now, run `pod install` and open the generated Xcode workspace. Open the precompiled header (`FRP-Prefix.pch`) and insert the following lines. These lines are semantically added to every file in the project.

```
1 // Pods
2 #import <ReactiveCocoa/ReactiveCocoa.h>
3 #import <500px-iOS-api/PXAPI.h>
4 #import <libextobjc/EXTScope.h>
5
6 // App Delegate
7 #import "FRPAppDelegate.h"
8 #define AppDelegate ((FRPAppDelegate *)[[UIApplication sharedApplication] delegat\
9 e])
```

Saul Mora, an opponent of relying on the app delegate singleton in this way, says every time that you do this, a puppy dies. But this is not a book about programming design patterns – it's a book about ReactiveCocoa, so we'll kill a few puppies.

Create a property on the app delegate to hold our 500px API client.

```
1 @property (nonatomic, readonly) PXAPIHelper *apiHelper;
```

Next, instantiate it in the `application: didFinishLaunchingWithOptions:` method.

```
1 self.apiHelper = [[PXAPIHelper alloc]
2     initWithHost:nil
3     consumerKey:@"DC2To2BS0ic1ChKDK15d44M42YHf9gbUJgdFoF0m"
4     consumerSecret:@"i8WL4chWoZ4kw9fh3jzHK7XzTer1y5tUNvsTFNnB"];
```

I'm providing a throwaway consumer key and secret – please don't go crazy with it. You can register your own application, too.

Alright. We're almost set up to do some loading. We'll need a *model* object to hold our information. I've created `FRPPhotoModel` below. It has an empty implementation.

```
@interface FRPPhotoModel : NSObject

@property (nonatomic, strong) NSString *photoName;
@property (nonatomic, strong) NSNumber *identifier;
@property (nonatomic, strong) NSString *photographerName;
@property (nonatomic, strong) NSNumber *rating;
@property (nonatomic, strong) NSString *thumbnailURL;
@property (nonatomic, strong) NSData *thumbnailData;
@property (nonatomic, strong) NSString *fullsizedURL;
@property (nonatomic, strong) NSData *fullsizedData;

@end
```

Great. Nearly there.

Instead of loading the content in our view controller directly, let's abstract that logic into another class. Create an `NSObject` subclass named `FRPPhotoImporter`.

None of this code has been particularly functional yet. Don't worry, we're getting there! The photo importer isn't going to actually return a `FRPPhotoModel` objects. Instead, it's going to return *signals* that carry with them the latest results from the API.

```
@interface FRPPhotoImporter : NSObject

+(RACSignal *)importPhotos;

@end
```

The photo importer's `importPhotos` method returns a `RACSignal` that sends the latest results from the API. This `RACSignal` will be a `RACReplaySubject` under the hood. But, since the ReactiveCocoa guidelines recommend *against* using `RACSubjects`, we declare the return type as a signal in the public header but a subject in our implementation. Let's take a look at it now.

```
+(RACReplaySubject *)importPhotos {
    RACReplaySubject *subject = [RACReplaySubject subject];

    NSURLRequest *request = [self popularURLRequest];

    [NSURLConnection
        sendAsynchronousRequest:request
        queue:[NSOperationQueue mainQueue]
        completionHandler:^(NSURLResponse *response,
            NSData *data, NSError *connectionError) {
```

```
11          if (data) {
12              id results = [NSJSONSerialization
13                  JSONObjectWithData:data options:0 error:nil];
14
15              [subject sendNext:[[[results[@"photos"] rac_sequence] map:
16                  ^id(NSDictionary *photoDictionary) {
17                  FRPPhotoModel *model = [FRPPhotoModel new];
18
19                  [self configurePhotoModel:model
20                      withDictionary:photoDictionary];
21                  [self downloadThumbnailForPhotoModel:model];
22
23                  return model;
24              }] array]];
25              [subject sendCompleted];
26          }
27          else {
28              [subject sendError:connectionError];
29          }
30      }];
31
32      return subject;
33  }
```

There's a lot going on here, so let's take it slow. First, we create a new `RACReplaySubject` instance. This will be our return value. Then, we're creating an `NSURLRequest` to access the popular photo models on 500px. We subsequently send that request asynchronously and then *immediately* return the subject. The immediacy of this return is important to note.

The subject was caught in the lexical scope of the asynchronous network request's callback block. When the data is returned from the API and the callback in invoked, then the subject will be sent values. Those values are going to be received by whoever subscribes to our subject.

This is a really common pattern you'll see when performing asynchronous operations.

1. Create a subject.
2. Send the subject values from within the completion block of the asynchronous call.
3. Immediately return the subject.

It's important to note the difference between a normal `RACSubject` and its subclass, `RACReplaySubject`. The replay variant ensures that the underlying subject is only subscribed to once, which prevents duplicate work from being performed (desirable in this case of network activity). The replay subject will store the values that are returned and forward them to new subscribers – perfect for our

needs. As ReactiveCocoa developer Justin Spahr-Summers pointed out, this prevents a possible race condition as well.

We're sending over a *completed* data set instead of a stream of single values over time. It would be "more reactive" if we instead sent a stream of individual photo models that could be concatenated later on. This would also help with pagination, but we're not going to address this pattern because it's a little more advanced. Check out octokit for an example of this concatenation.

The url request-constructing method looks like the following.

```
+(NSURLRequest *)popularURLRequest {
    return [AppDelegate.apiHelper urlRequestForPhotoFeature:PXAPIHelperPhotoFeatu\
rePopular resultsPerPage:100 page:0 photoSizes:PXPhotoModelSizeThumbnail sortOrde\
r:PXAPIHelperSortOrderRating except:PXPhotoModelCategoryNude];
}
```

What is the subject sending? Well, that depends on the callback block.

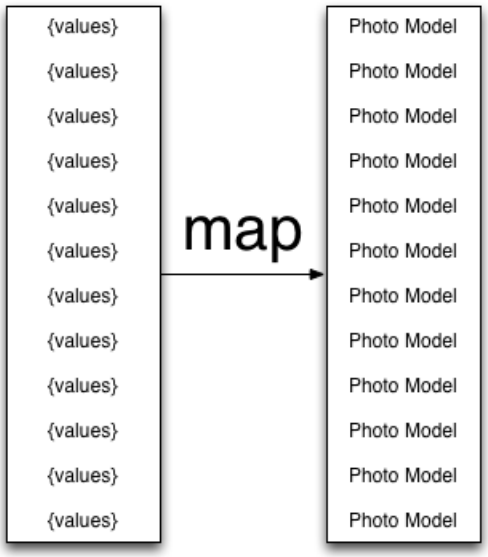
```
if (data) {
    id results = [NSJSONSerialization JSONObjectWithData:data options:0 error:nil\
];

    [subject sendNext:[[[results[@"photos"] rac_sequence]
        map:^id(NSDictionary *photoDictionary) {
        FRPPhotoModel *model = [FRPPhotoModel new];

        [self configurePhotoModel:model
            withDictionary:photoDictionary];
        [self downloadThumbnailForPhotoModel:model];

        return model;
    }] array]];
    [subject sendCompleted];
}
else {
    [subject sendError:connectionError];
}
```

We test whether or not data was returned. Arguably this isn't the best way to check for an error condition, but this is a pedagogical example. If the `data` is `nil`, then we send an error. Otherwise, we deserialize the JSON data process it. It's not immediately clear how we're doing this, so let's take a closer look.

```
[subject sendNext:[[[results[@"photos"] rac_sequence] map:^id(NSDictionary *photo\
Dictionary) {
    FRPPhotoModel *model = [FRPPhotoModel new];

    [self configurePhotoModel:model
        withDictionary:photoDictionary];
    [self downloadThumbnailForPhotoModel:model];

    return model;
}] array]];
[subject sendCompleted];
```

Deconstructing the rather terse (but typical-looking) first expression, we're sending a value along the subject. That value is the results' `photos` value, turned into a sequence, mapped, and then turned back into an array. This is a similar technique to what we saw about `map:` in the last chapter.

This mapping is very interesting. For each item in the sequence, a new `FRPPhotoModel` object is created, configured, and returned. This creates a corresponding array of photo model objects for each item in the `results[@"photos"]` array. This corresponding array of models is the value that's sent along the subject. Finally, we send the completed value to let any subscribers know that we're finished.



Note that the ability to manually send values along signals is atypical, belonging only to `RACSubject` instances.

The `configurePhotoModel:withDictionary:` method looks like the following.

```
+(void)configurePhotoModel:(FRPPhotoModel *)photoModel withDictionary:(NSDictiona\
ry *)dictionary {
    // Basics details fetched with the first, basic request
    photoModel.photoName = dictionary[@"name"];
    photoModel.identifier = dictionary[@"id"];
    photoModel.photographerName = dictionary[@"user"][@"username"];
    photoModel.rating = dictionary[@"rating"];

    photoModel.thumbnailURL = [self urlForImageSize:3 inArray:dictionary[@"images\
"]];

    // Extended attributes fetched with subsequent request
    if (dictionary[@"comments_count"]) {
        photoModel.fullsizedURL = [self urlForImageSize:4
            inArray:dictionary[@"images"]];
    }
}
```

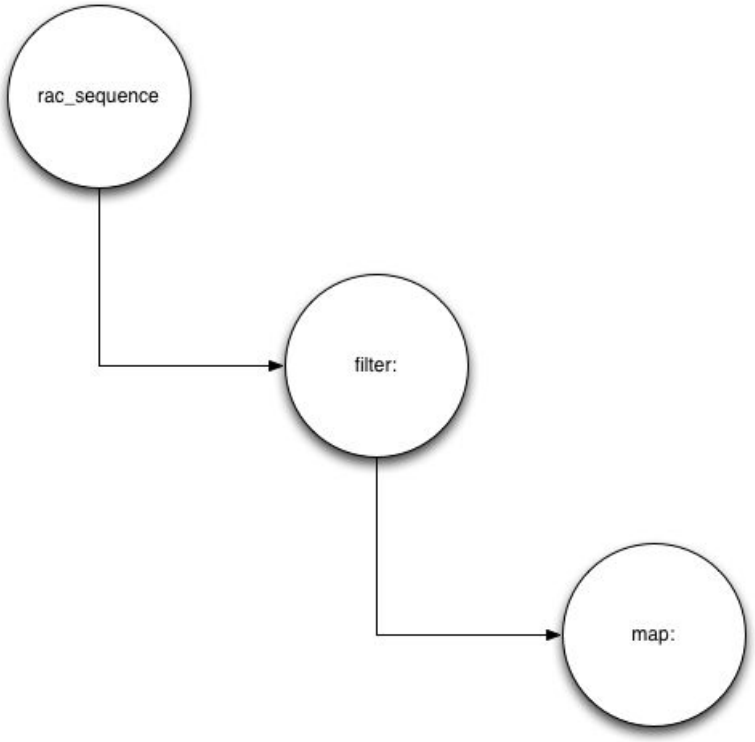
This is all pretty basic stuff except for setting the url properties. They rely on another method to extract the correct url from the list of images that the 500px API returns. The data is in the following format.

```
    (
        {
            size = size;
            url = ...;
        }
    );
```

So it's an array of dictionaries, each dictionary containing a size and a URL. Our method to suss it all out looks like the following.

```
+(NSString *)urlForImageSize:(NSInteger)size inDictionary:(NSArray *)array {

    return [[[[array rac_sequence] filter:^BOOL(NSDictionary *value) {
        return [value[@"size"] integerValue] == size;
    }] map:^id(id value) {
        return value[@"url"];
    }] array] firstObject];
}
```

There is some implicit error-handling going on here. The `firstObject` method on a `NSArray` returns `nil` if the sequence is empty. First, we filter out dictionaries whose size doesn't match our parameter. Then we use `map:` to extract the `url` value out of those dictionaries. Our result is a sequence of `NSString` objects, which we turn back into an array of strings, and return the first object.



This chaining of signals is very common in ReactiveCocoa. Values are pushed from `rac_sequence` to the `filter:` method, which finally pushes them to the `map:` method. Calling `array` on this result turns the resulting sequence into an array.

Finally, our `downloadThumbnailForPhotoModel:` method looks like the following.

```
+(void)downloadThumbnailForPhotoModel:(FRPPhotoModel *)photoModel {
    NSAssert(photoModel.thumbnailURL, @"Thumbnail URL must not be nil");

    NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL URLWithString:pho\
toModel.thumbnailURL]];
    [NSURLConnection sendAsynchronousRequest:request
        queue:[NSOperationQueue mainQueue]
        completionHandler:^(NSURLResponse *response,
            NSData *data, NSError *connectionError) {
        photoModel.thumbnailData = data;
    }];
```

```
}
```

There's nothing particularly reactive about this method – it just downloads data at the thumbnail's url and then, in the completion block, sets the appropriate property.

We've covered almost everything we need for the basic gallery – let's take a look at the view controller. In the implementation file, declare the following private property.

```
@interface FRPGalleryViewController ()

@property (nonatomic, strong) NSArray *photosArray;

@end
```

We've already covered the initializer, so let's take a look at `viewDidLoad`.

```
static NSString *CellIdentifier = @"Cell";

- (void)viewDidLoad
{
    [super viewDidLoad];

    // Configure self
    self.title = @"Popular on 500px";

    // Configure view
    [self.collectionView registerClass:[FRPCell class]
        forCellWithReuseIdentifier:CellIdentifier];

    // Reactive Stuff
    @weakify(self);
    [RACObserve(self, photosArray) subscribeNext:^(id x) {
        @strongify(self);
        [self.collectionView reloadData];
    }];

    // Load data
    [self loadPopularPhotos];
}
```

We set the title of the view controller and register a class with our collection view. The collection view is going to create or dequeue instances of this class for its cells. I'm referencing a `UICollectionViewCell` subclass that doesn't exist yet, here. We'll create it soon.

Under the heading of "Reactive Stuff", you'll find these curious statements.

```
@weakify(self);
[RACObserve(self, photosArray) subscribeNext:^(id x) {
    @strongify(self);
    [self.collectionView reloadData];
}];
```

`RACObserver` is a C macro that takes two parameters: an object and a key path on that object. It returns a signal whose values are sent whenever the key path's value changes. A completion value is sent when the object, in this case `self`, is deallocated. We subscribe to this signal in order to reload our collection view whenever our `photosArray` property is changed.

The weakify/strongify dance is all too common in Objective-C under ARC. Weakify creates a new, weak variable assigned to `self`. Strongify then creates a new, strong variable in its scope assigned to the weak self. When strongify does this, it's using what's called a "shadow variable" – so named because the new, strong variable is called `self`, replacing the former strong reference to `self`.

Basically, the `subscribeNext:` block is going to capture `self` in its lexical scope, causing a reference cycle between `self` and the block. The block is strongly referenced by the return value of `subscribeNext:`, a `RACSubscriber` instance. This is then captured by the `RACObserver` macro, which will be automatically deallocated once its first parameter, `self` is deallocated. Without the weakify/strongify invocations, `self` would never be deallocated.

Finally, we actually call `loadPopularPhotos`, which we'll cover next.

```
-(void)loadPopularPhotos {
    [[FRPPhotoImporter importPhotos] subscribeNext:^(id x) {
        self.photosArray = x;
    } error:^(NSError *error) {
        NSLog(@"Couldn't fetch photos from 500px: %@",
            error);
    }];
}
```

This method is responsible for actually calling the `importPhotos` method in our `FRPPhotoImporter` (add a `#import` for its header now). It subscribes to the results in order to set our private property. Beware that this introduces some state to our application. Unfortunately, due to the architecture of the `UICollectionViewDataSource` protocol, this is unavoidable.

Let's take a look at the those protocol methods now. There are two that are required, implemented below.

```
-(NSInteger)collectionView:(UICollectionView *)collectionView numberOfItemsInSect\
ion:(NSInteger)section {
    return self.photosArray.count;
}

-(UICollectionViewCell *)collectionView:(UICollectionView *)collectionView cellFo\
rItemAtIndexPath:(NSIndexPath *)indexPath {
    FRPCell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:CellId\
entifier forIndexPath:indexPath];

    [cell setPhotoModel:self.photosArray[indexPath.row]];

    return cell;
}
```

The first method simply returns the number of cells in the collection view – in this case, exactly the number of cells in the `photosArray` property. The next method dequeues a cell from the collection view and calls the `setPhotoModel:` method on that cell (we haven't implemented it yet, don't worry). This code should look very familiar if you've ever dealt with `UITableViewDataSource` methods.

That's the complete implementation of our view controller. Let's create that `UICollectionViewCell` subclass now, and call it `FRPCell`. Modify it's header file to match the following:

```
@class FRPPhotoModel;

@interface FRPCell : UICollectionViewCell

-(void)setPhotoModel:(FRPPhotoModel *)photoModel;

@end
```

And in the implementation file, add the following private interface.

```
#import "FRPPhotoModel.m"

@interface FRPCell ()

@property (nonatomic, weak) UIImageView *imageView;
@property (nonatomic, strong) RACDisposable *subscription;

@end
```

There are two properties here: an image view, and a subscription. The image view is a `weak` property, since it will belong to its superview (this is standard practice when writing `UICollectionViewCell` subclasses). We'll instantiate and set the image view later. The next property is a subscription, which we'll touch on when we use ReactiveCocoa to set the image view's `image` property. Note that this *must* be a strong property, not weak, or else you'll get a runtime exception.

The implementation of the cell looks like this.

```
- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (!self) return nil;

    // Configure self
    self.backgroundColor = [UIColor darkGrayColor];

    // Configure subivews
    UIImageView *imageView = [[UIImageView alloc]
        initWithFrame:self.bounds];
    imageView.autoresizingMask =
        UIViewAutoresizingFlexibleHeight |
        UIViewAutoresizingFlexibleWidth;
    [self.contentView addSubview:imageView];
    self.imageView = imageView;

    return self;
}
```

Standard `UICollectionViewCell` subclass boilerplate. It creates and assigns the `imageView` property. Note that we must store the image view in an intermediate, strong local variable so that it's not deallocated immediately after being assigned to the `imageView` property on `self`. You'll get a compiler warning, otherwise.

There are two other methods we must implement to complete our 500px gallery. The first is the `setPhotoModel:` method.

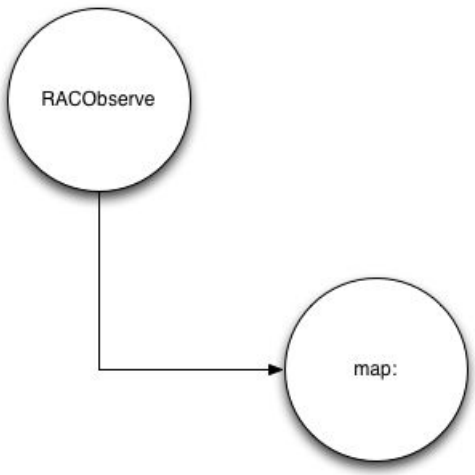
```
-(void)setPhotoModel:(FRPPhotoModel *)photoModel {
    self.subscription =
        [[[RACObserve(photoModel, thumbnailData)
            filter:^BOOL(id value) {
        return value != nil;
    }] map:^id(id value) {
        return [UIImage imageWithData:value];
    }] setKeyPath:@keypath(self.imageView, image)
        onObject:self.imageView];
}
```

This method assigns the `subscription` property we saw earlier. It assigns it to the return value of `setKeyPath:onObject:`. In practice, this method isn't used a whole lot; we use the `RAC` C macro instead. We'll cover that later, though.

The subscription is necessary for two reasons. First, we want to *dispose* of it later, making it not accept new values. Second, the signal its subscribing to is *cold*, so no work will be done until someone subscribes to it.

`setKeyPath:onObject:` is a method on `RACSignal` that binds the latest value of the signal to the key path of the object. In our case, we're calling this method on a chained signal. Let's take a closer look at it now.

```
[[RACObserve(photoModel, thumbnailData)
    filter:^BOOL(id value) {
    return return value != nil;
}] map:^id(id value) {
    return [UIImage imageWithData:value];
}]
```



The signal starts with the `RACObserve` C macro. This macro simply returns a signal which is sent new values whenever the specified key path of the target changes. In this case, our target is the photo model and the key path is that for its `thumbnailData` property. We're filtering those values that are `nil`. Finally, we map the filtered signal from `NSData` instances to `UIImage` ones.

Note that this mapping of `NSData` to `UIImage` only works on small images and can cause performance delays if done frequently or on large images. Ideally, we'd cache those decompressed images in memory instead of recomputing them each time. This technique is beyond the scope of the book, but the ReactiveCocoa technique we're about to see will work with either approach.

Note that the `thumbnailData` property doesn't need to be set at all for this to work – it can be set by another part of the application at a later time, and our cell's image view's image will be updated for us. Like magic!

Are we breaking Model-View-Controller a little bit here? A little bit, maybe. Luckily, we'll see in the next chapter a way around our MVC woes, so I'm not going to worry about this now.

The subscription will automatically be *disposed of* once the `onObject:` parameter is deallocated. Throwing a wrench in that plan, our cell instances are *reused* by the collection view, so we'll need to dispose of that subscription when the cell is about to be reused. We'll do that by overriding the following `UICollectionViewCell` method.

```
1  -(void)prepareForReuse {
2      [super prepareForReuse];
3
4      [self.subscription dispose], self.subscription = nil;
5  }
```

This method is called just before the cell is reused. If we ran the application now, we'd see the following results.

Great! We can scroll to demonstrate that our disposing of the subscription manually works.

Adding to FunctionalReactivePixels

A simple gallery view is fine, but what if we want to see a higher-resolution image? Let's create a new view controller and push it onto the navigation stack whenever a user taps a cell.

```
-(void)collectionView:(UICollectionView *)collectionView
    didSelectItemAtIndexPath:(NSIndexPath *)indexPath {
    FRPFullSizePhotoViewController *viewController =
        [[FRPFullSizePhotoViewController alloc]
            initWithPhotoModels:self.photosArray
              currentPhotoIndex:indexPath.item];
    viewController.delegate = self;
    [self.navigationController
        pushViewController:viewController animated:YES];
}
```

There's nothing special about this method – just plain Objective-C. Also, don't forget to `#import` the new view controller's header in this implementation file. Let's create that view controller now.

Create the `FRPFullSizePhotoViewController` class as a subclass of `UIViewController`. This isn't going to be a particularly reactive view controller; most of the work is actually just boilerplate to get a `UIPageViewController` child view controller working. The header file of our new view controller looks like the following:

```
@class FRPFullSizePhotoViewController;

@protocol FRPFullSizePhotoViewControllerDelegate <NSObject>

-(void)userDidScroll:(FRPFullSizePhotoViewController *)viewController toPhotoAtIn\
dex:(NSInteger)index;

@end

@interface FRPFullSizePhotoViewController : UIViewController

-(instancetype)initWithPhotoModels:(NSArray *)photoModelArray
    currentPhotoIndex:(NSInteger)photoIndex;

@property (nonatomic, readonly) NSArray *photoModelArray;
@property (nonatomic, weak)
    id<FRPFullSizePhotoViewControllerDelegate> delegate;

@end
```

Go back to the gallery view controller and implement the required delegate method.

```
-(void)userDidScroll:(FRPFullSizePhotoViewController *)viewController
    toPhotoAtIndex:(NSInteger)index {
    [self.collectionView scrollToItemAtIndexPath:
        [NSIndexPath indexPathForItem:index inSection:0]
        atScrollPosition:UICollectionViewScrollPositionCenteredVertically
        animated:NO];
}
```

This method will update the scroll position of our collection view whenever we swipe to a new photo in the full-sized photo view controller. That way, the photo the user was just looking at will be visible when they hit the back button.

`#import` the necessarily model file and add the following two private properties.

```
@interface FRPFullSizePhotoViewController () <UIPageViewControllerDataSource, UIP\
ageViewControllerDelegate>

// Private assignment
@property (nonatomic, strong) NSArray *photoModelArray;

// Private properties
@property (nonatomic, strong) UIPageViewController *pageViewController;

@end
```

The `photoModelArray` array is a public, *readonly* property, but a private *read/write* one. The second property is our child view controller.

Our initializer will look like this.

```
-(instancetype)initWithPhotoModels:(NSArray *)photoModelArray currentPhotoIndex:(\
NSInteger)photoIndex
{
    self = [self init];
    if (!self) return nil;

    // Initialized, read-only properties
    self.photoModelArray = photoModelArray;

    // Configure self
    self.title = [self.photoModelArray[photoIndex] photoName];

    // View controllers
```

```
    self.pageViewController = [[UIPageViewController alloc]
        initWithTransitionStyle:UIPageViewControllerTransitionStyleScroll
        navigationOrientation:UIPageViewControllerNavigationOrientationHorizontal
        options:@{UIPageViewControllerOptionInterPageSpacingKey: @(30)}];
    self.pageViewController.dataSource = self;
    self.pageViewController.delegate = self;
    [self addChildViewController:self.pageViewController];

    [self.pageViewController
        setViewControllers:@[[self
            photoViewControllerForIndex:photoIndex]]
        direction:UIPageViewControllerNavigationDirectionForward
        animated:NO
        completion:nil];

    return self;
}
```

We set our properties, configure our title to represent the current photo, and set up our `pageViewController`. All very boring. Our `viewDidLoad` method is similarly simple.

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    // Configure self's view
    self.view.backgroundColor = [UIColor blackColor];

    // Configure subviews
    self.pageViewController.view.frame = self.view.bounds;
    [self.view addSubview:self.pageViewController.view];
}
```

I should note that, in my app, I've disabled landscape orientations for the sake of brevity. This is not a book about autoresizing masks or Auto Layout. Go check out Erica Sadun's book on Auto Layout for more information.

Let's take a look at `UIPageViewController`'s data source and delegate protocol methods next.

```
- (void)pageViewController:(UIPageViewController *)pageViewController
    didFinishAnimating:(BOOL)finished
    previousViewControllers:(NSArray *)previousViewControllers
    transitionCompleted:(BOOL)completed {
    self.title = [[self.pageViewController.viewControllers.firstObject photoModel\
] photoName];
    [self.delegate userDidScroll:self toPhotoAtIndex:
        [self.pageViewController.viewControllers.firstObject photoIndex]];
}

- (UIViewController *)pageViewController:(UIPageViewController *)pageViewControll\
er
    viewControllerBeforeViewController:(FRPPhotoViewController *)viewController {
    return [self photoViewControllerForIndex:viewController.photoIndex - 1];
}

- (UIViewController *)pageViewController:(UIPageViewController *)pageViewControll\
er
    viewControllerAfterViewController:(FRPPhotoViewController *)viewController {
    return [self photoViewControllerForIndex:viewController.photoIndex + 1];
}
```

While there's nothing technically reactive about these methods, they do embody a certain sense of functional-ness. I admire the abstractions made for this particular type of view controller. Well done, Apple.

Our view controller-creating method looks like the following.

```
-(FRPPhotoViewController *)photoViewControllerForIndex:(NSInteger)index {
    if (index >= 0 && index < self.photoModelArray.count) {
        FRPPhotoModel *photoModel = self.photoModelArray[index];

        FRPPhotoViewController *photoViewController =
            [[FRPPhotoViewController alloc]
                initWithPhotoModel:photoModel index:index];
        return photoViewController;
    }

    // Index was out of bounds, return nil
    return nil;
}
```

It basically creates and configures an `FRPPhotoViewController`, a `UIViewController` subclass which we'll create now. Here's it header file.

```
@class FRPPhotoModel;

@interface FRPPhotoViewController : UIViewController

-(instancetype)initWithPhotoModel:(FRPPhotoModel *)photoModel index:(NSInteger)ph\
otoIndex;

@property (nonatomic, readonly) NSInteger photoIndex;
@property (nonatomic, readonly) FRPPhotoModel *photoModel;

@end
```

This view controller is going to be very straightforward: display a photo model's full-sized image, and cue the photo importer to download that image. It's so simple that I'll show you the entirety of its implementation now.

```
// Model
#import "FRPPhotoModel.h"

// Utilities
#import "FRPPhotoImporter.h"
#import <SVProgressHUD.h>

@interface FRPPhotoViewController ()

// Private assignment
@property (nonatomic, assign) NSInteger photoIndex;
@property (nonatomic, strong) FRPPhotoModel *photoModel;

// Private properties
@property (nonatomic, weak) UIImageView *imageView;

@end

@implementation FRPPhotoViewController

-(instancetype)initWithPhotoModel:(FRPPhotoModel *)photoModel
    index:(NSInteger)photoIndex
{
    self = [self init];
    if (!self) return nil;

```

```
27      self.photoModel = photoModel;
28      self.photoIndex = photoIndex;
29
30      return self;
31  }
32
33  -(void)viewDidLoad
34  {
35      [super viewDidLoad];
36
37      // Configure self's view
38      self.view.backgroundColor = [UIColor blackColor];
39
40      // Configure subviews
41      UIImageView *imageView = [[UIImageView alloc]
42          initWithFrame:self.view.bounds];
43      RAC(imageView, image) = [RACObserve(self.photoModel, fullsizedData)
44          map:^id(id value) {
45          return [UIImage imageWithData:value];
46      }];
47      imageView.contentMode = UIViewContentModeScaleAspectFit;
48      [self.view addSubview:imageView];
49      self.imageView = imageView;
50  }
51
52  -(void)viewWillAppear:(BOOL)animated {
53      [super viewWillAppear:animated];
54
55      [SVProgressHUD show];
56
57      // Fetch data
58      [[FRPPhotoImporter fetchPhotoDetails:self.photoModel]
59          subscribeError:^(NSError *error) {
60          [SVProgressHUD showErrorWithStatus:@"Error"];
61      } completed:^{
62          [SVProgressHUD dismiss];
63      }];
64  }
```

Like in our collection view cell, we're binding a `UIImageView`'s `image` property to be the mapped version of a model's data. The difference is our view controllers won't be reused, so we don't have to fuss with any subscription disposal – the subscription will be disposed of when the image view is deallocated.

The other interesting part of this implementation is the following in `viewWillAppear:`.

```
[SVProgressHUD show];

// Fetch data
[[FRPPhotoImporter fetchPhotoDetails:self.photoModel] subscribeError:^(NSError *e\
rror) {
    [SVProgressHUD showErrorWithStatus:@"Error"];
} completed:^{
    [SVProgressHUD dismiss];
}];
```

We should display the progress indicator to the user and dismiss it when we receive either an error or complete value. See, the 500px API will return a "short" photo model on the call to the popular API endpoint. We'll need some more details about the photo later on, so we'll need to call a second API endpoint, per-photo, to retrieve that information (including the full-size photo URL).

```
+(NSURLRequest *)photoURLRequest:(FRPPhotoModel *)photoModel {
    return [AppDelegate.apiHelper urlRequestForPhotoID:photoModel.identifier.inte\
gerValue];
}
```

We haven't implemented the `fetchPhotoDetails:` method yet, so let's return to the `FRPPhotoImport` now.

Declare the method in the header and then write its implementation below.

```
+(RACReplaySubject *)fetchPhotoDetails:(FRPPhotoModel *)photoModel {
    RACReplaySubject *subject = [RACReplaySubject subject];

    NSURLRequest *request = [self photoURLRequest:photoModel];
    [NSURLConnection sendAsynchronousRequest:request
        queue:[NSOperationQueue mainQueue]
        completionHandler:^(NSURLResponse *response,
        NSData *data, NSError *connectionError) {
        if (data) {
            id results = [NSJSONSerialization JSONObjectWithData:data options:0 e\
rror:nil][@"photo"];

            [self configurePhotoModel:photoModel
                withDictionary:results];
            [self
```

```
16                  downloadFullsizedImageForPhotoModel:photoModel];
17
18              [subject sendNext:photoModel];
19              [subject sendCompleted];
20          }
21          else {
22              [subject sendError:connectionError];
23          }
24      }];
25
26      return subject;
27  }
```

This method follows the same pattern we saw earlier with the `importPhotos` method. Our `downloadFullsizedImageForPhotoModel:` method is similarly similar to `downloadThumbnailForPhotoModel:`. What a prime suspect for some abstraction. Let's replace our thumbnail method, in addition to adding the following two.

```
1   +(void)downloadThumbnailForPhotoModel:(FRPPhotoModel *)photoModel {
2       [self download:photoModel.thumbnailURL
3           withCompletion:^(NSData *data) {
4           photoModel.thumbnailData = data;
5       }];
6   }
7
8   +(void)downloadFullsizedImageForPhotoModel:(FRPPhotoModel *)photoModel {
9       [self download:photoModel.fullsizedURL withCompletion:^(NSData *data) {
10          photoModel.fullsizedData = data;
11      }];
12  }
13
14  +(void)download:(NSString *)urlString
15      withCompletion:(void(^)(NSData *data))completion {
16      NSAssert(urlString, @"URL must not be nil");
17
18      NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL URLWithString:url\
19  String]];
20      [NSURLConnection sendAsynchronousRequest:request
21          queue:[NSOperationQueue mainQueue]
22          completionHandler:^(NSURLResponse *response, NSData *data, NSError *conne\
23  ctionError) {
24          if (completion) {
```

```
25              completion(data);
26          }
27      }];
28  }
```

I once worked with a client who subscribed to the opinion that if you're repeating more than two lines of code, it should be abstracted somehow. While I think that may be a bit extreme, I like that attitude.

Great. We can now run the app, tap on a photo, and see a full-size version of it. We can also swipe back and forth to see the next and previous photos. Good job!

Revisiting FunctionalReactivePixels

We hit a lot of the key parts of using ReactiveCocoa in the last section, but there are more opportunities to use ReactiveCocoa throughout the codebase. We're going to explore those opportunities now.

The first is in our gallery view controller. It's implementing the methods of three different protocols: the collection view data source, the collection view delegate, the full-sized photo view controller delegate.

We can abstract any implementations of delegate-type protocol methods (i.e.: those a `void` return type) away in instances of a class called `RACDelegateProxy`.

The delegate proxy is a "blank slate" of an object that you call `rac_signalForSelector:` on to get a signal that sends a new value whenever that selector is invoked.

Note: You *must* retain the delegate objects or else they will be released and you'll get an `EXC_BAD_ACCESS` exception. Add the following private property to the gallery view controller.

```
@property (nonatomic, strong) id collectionViewDelegate;
```

You also need to `#import` the file `RACDelegateProxy.h`, since it's not a core part of ReactiveCocoa.

Remove the `UICollectionViewDelegate` and `FRPFullSizePhotoViewControllerDelegate` methods. Add the following to `viewDidLoad`.

```
RACDelegateProxy *viewControllerDelegate = [[RACDelegateProxy alloc]
    initWithProtocol:@protocol(FRPFullSizePhotoViewControllerDelegate)];

[[viewControllerDelegate
    rac_signalForSelector:@selector(userDidScroll:toPhotoAtIndex:)
    fromProtocol:@protocol(FRPFullSizePhotoViewControllerDelegate)]
    subscribeNext:^(RACTuple *value) {
    @strongify(self);
    [self.collectionView scrollToItemAtIndexPath:
        [NSIndexPath indexPathForItem:[value.second integerValue] inSection:0]
        atScrollPosition:UICollectionViewScrollPositionCenteredVertically animate\
d:NO];
}];

self.collectionViewDelegate = [[RACDelegateProxy alloc]
    initWithProtocol:@protocol(UICollectionViewDelegate)];
[[self.collectionViewDelegate
    rac_signalForSelector:@selector(collectionView:didSelectItemAtIndexPath:)]
```

```
    subscribeNext:^(RACTuple *arguments) {
    @strongify(self);
    FRPFullSizePhotoViewController *viewController = [[FRPFullSizePhotoViewContro\
ller alloc]
        initWithPhotoModels:self.photosArray currentPhotoIndex:[(NSIndexPath *)ar\
guments.second item]];
    viewController.delegate =
        (id<FRPFullSizePhotoViewControllerDelegate>)viewControllerDelegate;
    [self.navigationController
        pushViewController:viewController animated:YES];
}];
```

We could also have called `rac_signalForSelector:` on `self`, using the same blocks. However, we would also need to provide empty stub methods in our view controller implementation in order to silence the compiler's "incomplete implementation" warning.

Next, we have one more opportunity for abstraction in this class. Our `loadPopularPhotos` method doesn't really do much except mutate our state. It would be awesome if ReactiveCocoa could somehow take care of that state mutation for us, so we don't have to worry about it. Luckily, I know just the thing.

We can remove the method and replace its invocation with the following lines of code in `viewDidLoad`.

```
RACSignal *photoSignal = [FRPPhotoImporter importPhotos];
RACSignal *photosLoaded = [photoSignal catch:^RACSignal *(NSError *error) {
    NSLog(@"Couldn't fetch photos from 500px: %@", error);
    return [RACSignal empty];
}];
RAC(self, photosArray) = photosLoaded;
[photosLoaded subscribeCompleted:^{
    @strongify(self);
    [self.collectionView reloadData];
}];
```

The first line is just our `importPhotos` invocation, except that we're storing its return value in a signal. Then, we "catch" errors on that signal and log them (like we were doing before, except with different syntax). The `catch:` method differs from `subscribeError:` subtly. It allows non-error values on the signal to "pass through" it, only invoking its block and sending along its return value in the event of an error. We'll use `catch:` here so that non-error values pass through. The `catch:` block just returns an empty signal. See this StackOverflow question for more details.

This pollutes our local variable scope a little bit, and can be even more concise with the following, equivalent method.

```
RAC(self, photosArray) = [[[[FRPPhotoImporter
    importPhotos]
    doCompleted:^{
        @strongify(self);
        [self.collectionView reloadData];
    }] logError] catchTo:[RACSignal empty]];
```

Using the `RAC` macro, we create a one-way binding from the latest values in the `photosLoaded` signal to the `photoArray` property. Awesome! State taken care of.

Let's take another look at our collection view cell subclass' implementation.

```
@interface FRPCell ()

@property (nonatomic, weak) UIImageView *imageView;

@property (nonatomic, strong) RACDisposable *subscription;

@end

@implementation FRPCell

- (id)initWithFrame:(CGRect)frame
{
   ...
}

-(void)prepareForReuse {
    [super prepareForReuse];

    [self.subscription dispose], self.subscription = nil;
}

-(void)setPhotoModel:(FRPPhotoModel *)photoModel {
    self.subscription = [[[RACObserve(photoModel, thumbnailData) filter:^BOOL(id \
value) {
        return value != nil;
    }] map:^id(id value) {
        return [UIImage imageWithData:value];
    }] setKeyPath:@keypath(self.imageView, image) onObject:self.imageView];
}

@end
```

There are two flags here that indicate an opportunity to abstract things by using ReactiveCocoa: one, we have state (the `subscription` property), and two, we are manually handling the `RACDisposable`'s lifetime. Any time you manually call `dispose` on a `RACDisposable` object, it's a good sign there's a "more reactive" way to do something. In our case, this is true.

We can abstract away the need to use the `prepareForReuse` method altogether by replacing creating a new property on `FRPCell`. This property is the `photoModel` (we were behaving before like it was a write-only property – now it's going to be readwrite). Place this property in the header file.

```
@property (nonatomic, strong) FRPPhotoModel *photoModel;
```

Next, we'll get rid of our `setPhotoModel:` method completely. Instead, we'll observe our own key path for our photo model's `thumbnailData`. Add this line to the cell's initializer.

```
RAC(self.imageView, image) = [[RACObserve(self, photoModel.thumbnailData)
    ignore:nil] map:^(NSData *data) {
    return [UIImage imageWithData:data];
}];
```

Notice how we're observing the `photoModel.thumbnailData` key path on `self`, and not the `thumbnailData` key path on `self.photoModel`. A subtle, but important distinction. The key path `photoModel.thumbnailData` will have a KVO notification triggered when the `photoModel` property on `self`, *or* the `photoModel`'s `thumbnailData` property changes.

And now we can completely get rid of the `subscription` property.

Network Layer Revisited

There is one more opportunity to further embrace the functional reactive programming philosophy, and that is in our networking layer, the `FRPPhotoImporter`. Let's take a look at our image downloading methods.

```
+(void)downloadThumbnailForPhotoModel:(FRPPhotoModel *)photoModel {
    [self download:photoModel.thumbnailURL withCompletion:^(NSData *data) {
        photoModel.thumbnailData = data;
    }];
}

+(void)downloadFullsizedImageForPhotoModel:(FRPPhotoModel *)photoModel {
    [self download:photoModel.fullsizedURL withCompletion:^(NSData *data) {
        photoModel.fullsizedData = data;
    }];
```

```
}

+(void)download:(NSString *)urlString withCompletion:(void(^)(NSData *data))compl\
etion {
    NSAssert(urlString, @"URL must not be nil");

    NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL URLWithString:url\
String]];
    [NSURLConnection sendAsynchronousRequest:request
        queue:[NSOperationQueue mainQueue]
        completionHandler:
            ^(NSURLResponse *response, NSData *data, NSError *connectionError) {
        if (completion) {
            completion(data);
        }
    }];
}
```

Completion blocks? Here's another opportunity to use signals. Further more, we can use ReactiveCocoa's `NSURLConnection` extension. Let's rewrite the methods to look like the following.

```
+(void)downloadThumbnailForPhotoModel:(FRPPhotoModel *)photoModel {
    RAC(photoModel, thumbnailData) = [self download:photoModel.thumbnailURL];
}

+(void)downloadFullsizedImageForPhotoModel:(FRPPhotoModel *)photoModel {
    RAC(photoModel, fullsizedData) = [self
        download:photoModel.fullsizedURL];
}

+(RACSignal *)download:(NSString *)urlString {
    NSAssert(urlString, @"URL must not be nil");

    NSURLRequest *request = [NSURLRequest
        requestWithURL:[NSURL URLWithString:urlString]];

    return [[[NSURLConnection rac_sendAsynchronousRequest:request]
        map:^id(RACTuple *value) {
        return [value second];
    }] deliverOn:[RACScheduler mainThreadScheduler]];
}
```

There are two big differences. First, we're using `RAC` to bind to the latest value of the signal returned by `downloadFullsizedImageForPhotoModel:`. (More on that in a moment.) Second, we're returning `NSURLConnection`'s `rac_sendAsynchronousRequest:` return value, mapped. Let's investigate what's going on here.

Reading the documentation, `rac_sendAsynchronousRequest:` returns a signal that sends values from the network request's response. The `RACTuple` it sends contains the response and the data, respectively. It'll error out when there's a network error. Finally, we change the *scheduler* associated with the signal to deliver on the main thread scheduler. A scheduler is analogous to a thread.

See, the network signal is going to return its values on a background scheduler. If we allow it to pass straight through, it might eventually bubble up to the UI, which should never be changed on a background thread.

Let's return to the two methods at the beginning. We have lines that look like the following.

```
RAC(photoModel, thumbnailData) =
    [self download:photoModel.thumbnailURL];
```

Normally, I wouldn't recommend binding a model to some signal. However, we know this signal is going to complete immediately after the network call completes, ending the subscription. This is safe as long as we only bind this key path once per instance.

We can also abstract away our use of the `RACReplaySubject` in a similar fashion. Let's revisit our `fetchPhotoDetails:` method.

```
+(RACReplaySubject *)fetchPhotoDetails:(FRPPhotoModel *)photoModel {
    RACReplaySubject *subject = [RACReplaySubject subject];

    NSURLRequest *request = [self photoURLRequest:photoModel];
    [NSURLConnection sendAsynchronousRequest:request
        queue:[NSOperationQueue mainQueue]
        completionHandler:
            ^(NSURLResponse *response, NSData *data, NSError *connectionError) {
        if (data) {
            id results =
                [NSJSONSerialization JSONObjectWithData:data options:0 error:nil]\
[@"photo"];

            [self configurePhotoModel:photoModel withDictionary:results];
            [self
                downloadFullsizedImageForPhotoModel:photoModel];

            [subject sendNext:photoModel];
```

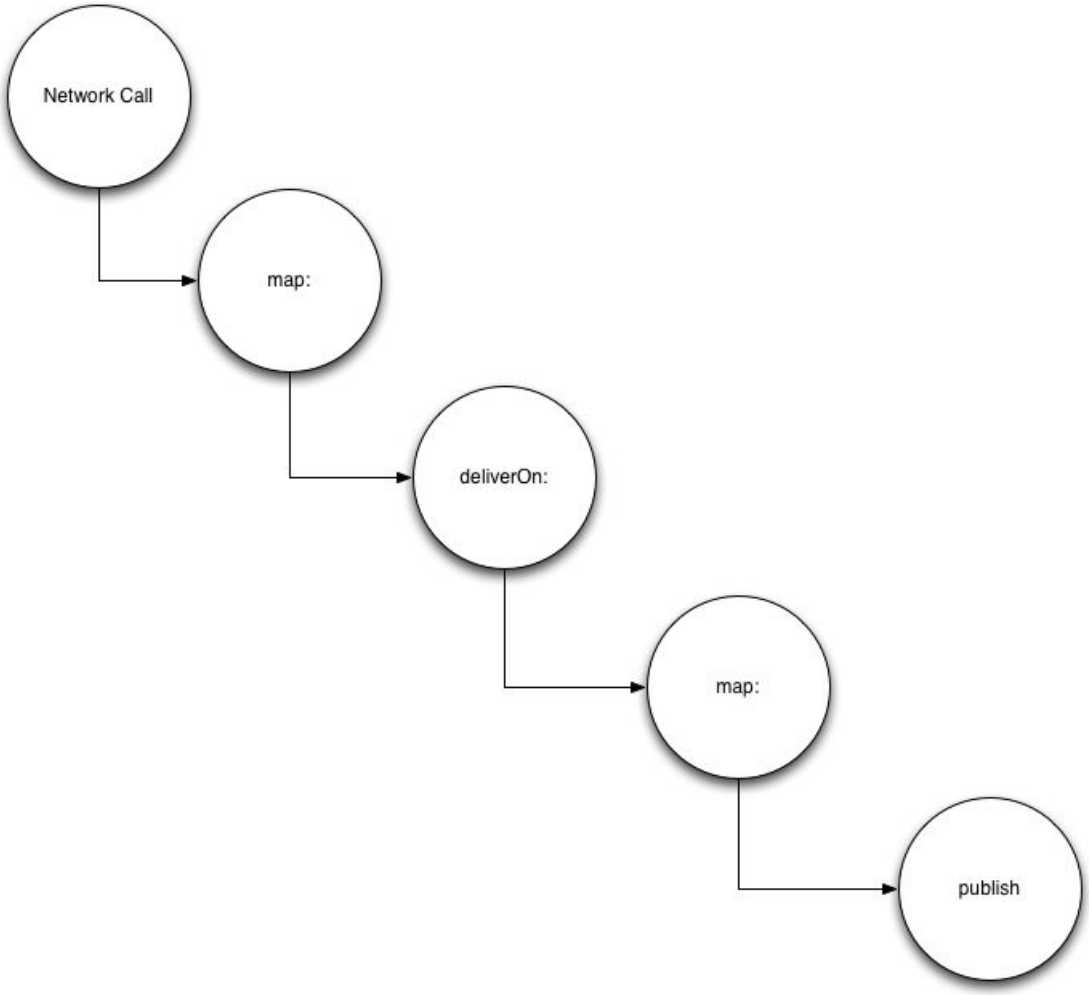
```
            [subject sendCompleted];
        }
        else {
            [subject sendError:connectionError];
        }
    }];

    return subject;
}
```

A little messy. Let's tidy it up a bit.

```
+(RACSignal *)fetchPhotoDetails:(FRPPhotoModel *)photoModel {
    NSURLRequest *request = [self photoURLRequest:photoModel];
    return [[[[[NSURLConnection
        rac_sendAsynchronousRequest:request] map:^id(RACTuple *value) {
        return [value second];
    }]
        deliverOn:[RACScheduler mainThreadScheduler]] map:^id(NSData *data) {
        id results =
            [NSJSONSerialization JSONObjectWithData:data options:0 error:nil][@"p\
hoto"];

        [self configurePhotoModel:photoModel withDictionary:results];
        [self downloadFullsizedImageForPhotoModel:photoModel];

        return photoModel;
    }] publish] autoconnect];
}
```

Note: The return value has changed from a `RACReplaySubject` to a `RACSignal`.

There's a lot to suss out here, so I've gone ahead and made a diagram.



We already saw how `deliverOn:` works, so let's focus on the last signal operation in our chain: `publish`. `publish` returns a `RACMulitcastConnection`, which will subscribe to the receiving signal when it's *connected*. That's what `autoconnect` does for us: connects to the underly signal when the signal it returns is subscribed to.

The signal that we return is going is *cold*, performing the fetch for each subscription, *at the time of subscription*. Because we're multicasting the underlying signal, that network request is only performed once, and its results multicasted. The result is a network signal that will only ever be performed once (when subscribed to), is cold (not being performed *until* subscribed to), and even cancellable (if the disposable generated from the subscription is disposed of).

Basically, as long as we could *guarantee* that the signal would only be subscribed to once, we wouldn't need `replay`.

Note that we can return the first `map:`, which uses a `RACTuple`, with the following `reduceEach:`, which gives us compile-time checking.

```
reduceEach:^id(NSURLResponse *response, NSData *data){
    return data;
}]
```

The remaining network-accessing, `importPhotos` method is similarly refactored.

```
+(RACSignal *)importPhotos {
    NSURLRequest *request = [self popularURLRequest];

    return [[[[[NSURLConnection
        rac_sendAsynchronousRequest:request]
        reduceEach:^id(NSURLResponse *response, NSData *data){
    return data;
}] deliverOn:[RACScheduler mainThreadScheduler]] map:^id(NSData *data) {
        id results =
            [NSJSONSerialization JSONObjectWithData:data options:0 error:nil];

        return [[[results[@"photos"] rac_sequence] map:^id(NSDictionary *photoDic\
tionary) {
            FRPPhotoModel *model = [FRPPhotoModel new];

            [self configurePhotoModel:model withDictionary:photoDictionary];
            [self downloadThumbnailForPhotoModel:model];

            return model;
        }] array];
    }] publish] autoconnect];
}
```

Conclusion

We've covered a lot of practical uses of ReactiveCocoa in this chapter. Here are some key take-aways.

Functional Programming Approaches Work Everywhere

As we saw in our data import code, we can use `map:` and `filter:` to help, even with non-reactive code. Always think in terms of abstractions and never the actual implementations.

Use `subscribeNext:` for Side-Effects

`subscribeNext:` and its family of methods for subscribing to side-effects of signals return `RACDisposable` instances which will hand around until the signal completes (usually upon deallocation). Use these methods for side-effects – things like interacting with the outside, non-reactive world.

Avoid Explicit State and Subscription Disposal

As per the Design Guidelines, avoid explicit subscription disposal whenever possible. Remember how we used `takeUntil:` to automatically dispose of a subscription in the `FRPCell` class. Using `takeUntil:` allows signal values to pass through until its parameter sends next or completed values itself, basically completing the receiver once that happens.

Memory Management is Magic

Under ARC, you ostensibly are freed from managing memory. So, too in ReactiveCocoa. The only caveat is not to capture `self` inside of any signal blocks.

That's it. We're all set to move on to Chapter 5, where we'll introduce the Model-View-ViewModel paradigm, add a log in mechanic to the app, and write some unit tests. Let's go!

Model View View-Model on iOS

There is a Zen Buddhist concept known as "beginner's mind." Zen teacher Shunryu Suzuki wrote "in the beginner's mind there are many possibilities, in the expert's mind there are few." Over the course of researching this book, I've often returned to this concept in order to re-centre myself and remind myself not to jump to conclusions about what seems new and uncomfortable.

In that spirit, I'd invite you to harken back to when you first began writing iOS applications. While the Model-View-Controller – MVC – paradigm might seem like the only way to write iOS applications *now*, then, you didn't know that. Your mind was open to infinite possibilities. Elders in the community guided you to MVC, since that's what they knew and what is recommended by Apple.

If you've been developing iOS apps for a while, you may be familiar with the alternative meaning behind MVC: Massive View Controller. A lot of the time, it's convenient to put business logic and other code into view controllers, even if that's not architecturally the most sound place to put it.

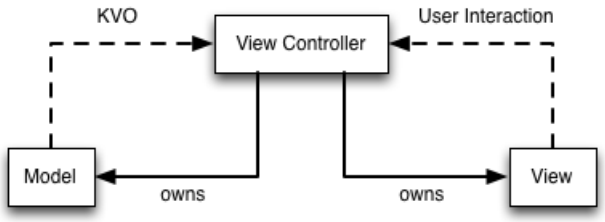
Model View View-Model, or MVVM, is an alternative to MVC that comes from Microsoft. I know, I know, the iOS community hasn't historically been huge fans of Microsoft, but their software engineering group does a great job. MVVM doesn't have to be used on the .Net platform – we can use it on iOS, too. As we'll see in this chapter, MVVM is incredibly applicable on iOS and goes very well with ReactiveCocoa. Use of MVVM helps reduce the amount of business logic in view controllers, which helps reduce their bloated size and also makes that business logic more testable.

What is MVVM?

In traditional MVC applications, you have three components: the model, the view, and the controller. The model holds your data while the view presents it. The controller mediates all interactions between the other two components.

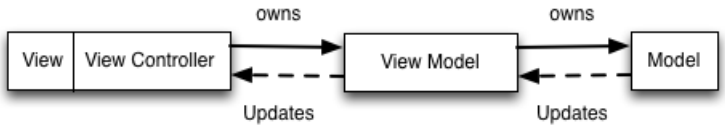
This mediation is important. Models shouldn't be aware of views and vice versa. Everything goes through the view controller. In typical iOS applications, the models are very "thin", meaning they don't contain business logic. The view is owned by UIKit, so its business logic has been (hopefully) tested by Apple. That leaves the view controller, which is seldom tested by unit tests.

When new data arrives, the model notifies the view controller, usually by key-value observation, and the view controller updates the view. When the view is interacted with, the view controller updates the model.



Typical MVC paradigm

As you can see, the view controller is implicitly responsible for a lot of things. Validating input, mapping model data to user-facing information, manipulating view hierarchies, etc. MVVM takes a lot of that logic away from the view controller.



MVVM high level

In MVVM, we tend to consider the view and the view controller as one entity (which explains why it's not called MVVCVM). Instead of the view controller performing a lot of those mediations between model and view, the view model does, instead.

We have "updates" in our MVVM diagram as somewhat ambiguous. There's nothing about MVVM that forces you to use specific mechanisms to update the view model, or the view. However, in the scope of this book, we'll be using ReactiveCocoa.

ReactiveCocoa will monitor for changes in the model and map those changes to properties on the view model, performing any necessary business logic.

As a concrete example, imagine our model contains a date called `dateAdded` that we want to monitor for changes, and update our view model's `dateAdded` property. The model's property is an `NSDate` instance, while the view model's is a transformed `NSString`. The binding would look something like the following (inside the view model's init method).

```
1 RAC(self, dateAdded) = [RACObserve(self.model, dateAdded) map:^(NSDate *date) {
2     return [[ViewModel dateFormatter] stringFromDate:date];
3 }];
```

`dateFormatter` is a class method on `ViewModel` that caches an `NSDateFormatter` instance so it can be reused (they're expensive to create). Then, the view controller can monitor the view model's `dateAdded` property, binding it to a label.

```
RAC(self.label, text) = RACObserve(self.viewModel, dateAdded);
```

We've now abstracted the logic of transforming the date to a string into our view model, where we might write unit tests for it. It seems contrived in this trivial example, but as we'll see, it helps reduce the amount of logic in your view controllers significantly.

Revisiting Functional Reactive Pixels

Before we continue to see how MVVM can be used to refactor our Functional Reactive Pixels demo, we need to do some housekeeping. The way we were using the 500px iOS SDK isn't compatible with their login system.

We're going to remove the `apiHelper` property from the header of our app delegate and replace the initializing line of code in the implementation file with the following, substituting your consumer key and secret as necessary.

```
[PXRequest setConsumerKey:consumerKey consumerSecret:consumerSecret];
```

Now, anywhere we were calling `AppDelegate.apiHelper` to create requests to the 500px API need to be replaced with `[PXRequest apiHelper]`.

Finally, please update the version number of the 500px iOS SDK in your CocoaPods file to `1.0.5`.

MVVM in Practice

The remainder of this chapter is going to deal with migrating the remaining code in the Functional Reactive Pixels demo to use MVVM. We're going to start by adding a new library to our podfile. The great folks at GitHub who wrote ReactiveCocoa have also written a base class to be used as a view model; it's called `ReactiveViewModel` and we're going to be using version `0.1.1`. Run `pod install` once you've updated your podfile.

The first class we're going to work on is the full sized photo view controller. We're starting here because it doesn't have a lot of business logic present to abstract to a view model. We're going to take things slow.

Currently, our `FRPFullSizePhotoViewController` contains an array of photo models and a current photo index. We're going to abstract both of these things away into our view model.

Remove the custom initializer from the header file and add a forward declaration for `FRPFullSizePhotoViewModel`. Then add a property for this new class.

```
@property (nonatomic, strong) FRPFullSizePhotoViewModel *viewModel;
```

In your implementation file, `#import` the new view model (don't worry, we'll create it shortly).

```
#import "FRPFullSizePhotoViewModel.h"
```

Next, remove the `photoModelArray` private property declaration. Rewrite our initializer to remove references to the `photoModelArray`. It should look like the following.

```
-(instancetype)init
{
    self = [super init];
    if (!self) return nil;

    // View controllers
    self.pageViewController = [[UIPageViewController alloc]
        initWithTransitionStyle:
            UIPageViewControllerTransitionStyleScroll
        navigationOrientation:
            UIPageViewControllerNavigationOrientationHorizontal
        options:
            @{UIPageViewControllerOptionInterPageSpacingKey: @(30)}];
    self.pageViewController.dataSource = self;
    self.pageViewController.delegate = self;
    [self addChildViewController:self.pageViewController];

    return self;
}
```

Add the following lines of code to your `viewDidLoad` implementation.

```
  // Configure child view controllers
[self.pageViewController setViewControllers:@[[self photoViewControllerForIndex:
    self.viewModel.initialPhotoIndex]]
    direction:UIPageViewControllerNavigationDirectionForward
    animated:NO completion:nil];

// Configure self
self.title = [self.viewModel.initialPhotoModel photoName];
```

We're referring to methods that we're about to write to give you a flavour for what goes within the view model. Finally, go to your `photoViewControllerForIndex` method. It's referencing the defunct `photoModelArray`. Replace it with the following implementation.

```
-(FRPPhotoViewController *)photoViewControllerForIndex:(NSInteger)index {
    if (index >= 0 && index < self.viewModel.photoArray.count) {
        FRPPhotoModel *photoModel = self.viewModel.model[index];

        FRPPhotoViewController *photoViewController =
            [[FRPPhotoViewController alloc] initWithPhotoModel:photoModel index:i\
ndex];
        return photoViewController;
    }

    // Index was out of bounds, return nil
    return nil;
}
```

Great. Now onto our view model itself. Create a new file named `FRPFullSizedPhotoViewModel`, a subclass of `RVMViewModel`. Based on the information it's going to encapsulate, as well as the methods we need in the accompanying view controller's implementation, we know that our header will look like the following.

```
@class FRPPhotoModel;

@interface FRPFullSizePhotoViewModel : RVMViewModel

-(instancetype)initWithPhotoArray:(NSArray *)photoArray
    initialPhotoIndex:(NSInteger)initialPhotoIndex;
-(FRPPhotoModel *)photoModelAtIndex:(NSInteger)index;

@property (nonatomic, readonly, strong) NSArray *model;
@property (nonatomic, readonly) NSInteger initialPhotoIndex;

@property (nonatomic, readonly) NSString *initialPhotoName;

@end
```

The `model` property is defined in `RVMViewModel` as `id`, so we redefined it as an `NSArray`. We also hang onto our initial photo index, and define a readonly property for our initial photo name. This is trivial logic we could place in our view controller, but we'll see more complex examples soon.

Next we'll need to write our implementation. First thing is first, we'll need to `#import` the `FRPPhotoModel` class' header file. Then we'll set up our private properties for readwrite access.

```
// Model
#import "FRPPhotoModel.h"

@interface FRPFullSizePhotoViewModel ()

// Private access
@property (nonatomic, assign) NSInteger initialPhotoIndex;

@end
```

Great! Next our initializer.

```
-(instancetype)initWithPhotoArray:(NSArray *)photoArray
    initialPhotoIndex:(NSInteger)initialPhotoIndex {
    self = [self initWithModel:photoArray];
    if (!self) return nil;

    self.initialPhotoIndex = initialPhotoIndex;

    return self;
}
```

The initializer passes along the photo model to its super implementation of `initWithModel:`, then sets the `initialPhotoIndex` itself. Our logic for the remaining two, readonly property getters is almost trivial.

```
-(NSString *)initialPhotoName {
    return [[self photoModelAtIndex:self.initialPhotoIndex] photoName];
}

-(FRPPhotoModel *)photoModelAtIndex:(NSInteger)index {
    if (index < 0 || index > self.model.count - 1) {
        // Index was out of bounds, return nil
        return nil;
    } else {
        return self.model[index];
    }
}
```

One relies on the other so we're not repeating logic. It also makes this very testable.

Finally, we need to set the view model on the full sized view controller; if we don't, it will be nil and nothing will appear on the screen. Navigate to the gallery view controller where we instantiate and push our full-sized view controller. Replace the logic with the following.

```
[[self rac_signalForSelector:
    @selector(collectionView:didSelectItemAtIndexPath:)
    fromProtocol:@protocol(UICollectionViewDelegate)]
    subscribeNext:^(RACTuple *arguments) {
    @strongify(self);

    NSIndexPath *indexPath = arguments.second;
    FRPFullSizePhotoViewModel *viewModel = [[FRPFullSizePhotoViewModel alloc]
        initWithPhotoArray:self.viewModel.model initialPhotoIndex:indexPath.item];

    FRPFullSizePhotoViewController *viewController =
        [[FRPFullSizePhotoViewController alloc] init];
    viewController.viewModel = viewModel;
    viewController.delegate = (id<FRPFullSizePhotoViewControllerDelegate>)self;
    [self.navigationController pushViewController:viewController animated:YES];
}];
```

And that's it!

We're not going to write tests for this view model until the next section, where we'll see how to apply the concepts of test-driven development to view models.

Let's go over our `FRPGalleryViewModel` now, since it's very basic. The logic that we want to abstract away from the view controller is loading the contents of the model from the API. Let's take a look at what that looks like.

```
@interface FRPGalleryViewModel : RVMViewModel

@property (nonatomic, readonly, strong) NSArray *model;

@end
```

Basic interface, declaring the model as an array. Next we have the simple implementation.

```
// Utilities
#import "FRPPhotoImporter.h"

@interface FRPGalleryViewModel ()

@end

@implementation FRPGalleryViewModel

```

```
-(instancetype)init {
    self = [super init];
    if (!self) return nil;

    RAC(self, model) = [[[FRPPhotoImporter importPhotos]
      logError] catchTo:[RACSignal empty]];

    return self;
}

@end
```

It's debatable whether or not we should place the logic for loading from the API in the initializer, or when the view model becomes *active*. We'll talk more about activating shortly, but I wanted to show how simple your view model can be. The migration of logic to the gallery view model instead of loading the contents directly from the view controller is very straightforward in this case. Initialize the view model in the view controller's initializer. Then, anywhere that was referring to the view controller's `self.model`, refer to `self.viewModel.model` instead.

We could go further with the view model, even abstracting away access to the model through a series of accessors, but that's excessive for this example. The important thing to note is that you can put as much or as little logic into your view models as you're comfortable with. I've found, personally, that the more I use this paradigm, the more and more logic I'm abstracting away. This means smaller view controllers and more cohesive, testable code.

Let's do one more example transitioning to view models before moving on to testing them.

Our last example is going to be the `FRPPhotoViewModel` that accompanies the `FRPPhotoViewController`. Create the view model subclass of `RVMViewModel` and move onto the view controller (we'll come back to the view model shortly).

Our new initializer for the view controller looks like the following.

```
-(instancetype)initWithViewModel:(FRPPhotoViewModel *)viewModel
    index:(NSInteger)photoIndex
{
    self = [self init];
    if (!self) return nil;

    self.viewModel = viewModel;
    self.photoIndex = photoIndex;

    return self;
}
```

Make sure to `#import` the necessary header file and declare a private property for the view model.

Now, we need to initialize the view controllers with the new init method. Take a look at the `photoViewControllerForIndex:` method that vends view controllers to the page view controller.

```
-(FRPPhotoViewController *)photoViewControllerForIndex:(NSInteger)index {
    FRPPhotoModel *photoModel = [self.viewModel photoModelAtIndex:index];
    if (photoModel) {
        FRPPhotoViewModel *photoViewModel =
        [[FRPPhotoViewModel alloc] initWithModel:photoModel];
        FRPPhotoViewController *photoViewController =
        [[FRPPhotoViewController alloc]
            initWithViewModel:photoViewModel index:index];
        return photoViewController;
    }

    return nil;
}
```

We're creating a view model to pass to our new initializer.

In our `viewDidLoad` method, we're going to use our new view model to provide data to our image view, as well as displaying and showing the progress HUD to indicate to the user that the image is being downloaded. The problem is that the image loading, being business logic, belongs in the view model, but the logic to *initiate* that download when the view appears should *not* be present in the view model – remember that good view models do not reference the view itself *at all*. So how do we mix these two pieces of logic?

The answer is that we use the *active* state of the view model. `RVMViewModel` provides an `active` boolean property which can be set when the view controller becomes "active" – whatever that means in the context. In our context, we'll use `viewWillAppear:` and `viewDidDisappear:` methods to set this property.

```
-(void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

    self.viewModel.active = YES;
}

-(void)viewDidDisappear:(BOOL)animated {
    [super viewDidDisappear:animated];

    self.viewModel.active = NO;
}
```

Fairly straightforward. Let's take a look at our new `viewDidLoad` method.

```
-(void)viewDidLoad {
    [super viewDidLoad];

    // Configure self's view
    self.view.backgroundColor = [UIColor blackColor];

    // Configure subviews
    UIImageView *imageView = [[UIImageView alloc] initWithFrame:self.view.bounds];
    RAC(imageView, image) = RACObserve(self.viewModel, photoImage);
    imageView.contentMode = UIViewContentModeScaleAspectFit;
    [self.view addSubview:imageView];
    self.imageView = imageView;

    [RACObserve(self.viewModel, loading) subscribeNext:^(NSNumber *loading){
        if (loading.boolValue) {
            [SVProgressHUD show];
        } else {
            [SVProgressHUD dismiss];
        }
    }];
}
```

The image view's image property binding is standard ReactiveCocoa. The interesting bits are underneath when we're using our `loading`. We show our progress HUD when the loading signal sends `YES` and dismiss it when loading signal sends `NO`. We'll see how the loading signal itself depends on the `didBecomeActiveSignal`. Now it's just up to the view model to kick off a network request for that photo's image data.

Our interface declaration looks like the following:

```
@class FRPPhotoModel;

@interface FRPPhotoViewModel : RVMViewModel

@property (nonatomic, readonly) FRPPhotoModel *model;

@property (nonatomic, readonly) UIImage *photoImage;
@property (nonatomic, readonly, getter = isLoading) BOOL loading;

-(NSString *)photoName;

@end
```

The `model` and `photoImage` properties' uses have already been explained. The `photoName` *de facto* property is used elsewhere in the code base to set things like the paging view controller's title. You can check out the GitHub repository (referenced below) for more details. Let's take a look at the implementation.

```
#import "FRPPhotoViewModel.h"

//Utilities
#import "FRPPhotoImporter.h"
#import "FRPPhotoModel.h"

@interface FRPPhotoViewModel ()

@property (nonatomic, strong) UIImage *photoImage;
@property (nonatomic, assign, getter = isLoading) BOOL loading;

@end

@implementation FRPPhotoViewModel

-(instancetype)initWithModel:(FRPPhotoModel *)photoModel {
    self = [super initWithModel:photoModel];
    if (!self) return nil;

    @weakify(self);
    [self.didBecomeActiveSignal subscribeNext:^(id x) {
        @strongify(self);
        self.loading = YES;
        [[FRPPhotoImporter fetchPhotoDetails:self.model]
            subscribeError:^(NSError *error) {
            NSLog(@"Could not fetch photo details: %@", error);
        } completed:^{
            self.loading = NO;
            NSLog(@"Fetched photo details.");
        }];
    }];

    RAC(self, photoImage) =
        [RACObserve(self.model, fullsizedData) map:^id(id value) {
        return [UIImage imageWithData:value];
    }];

    return self;
```

```
}

-(NSString *)photoName {
    return self.model.photoName;
}
```

The `didBecomeActiveSignal` is subscribed to with a side-effect of downloading the the photo details, including its full-sized image data. Then, the `photoImage` property is bound to a mapped version of the model's.

This approach of using the `didBecomeActiveSignal` to initiate expensive tasks like network operations is far preferable to our earlier example of starting them in the initializer method.

While that's all we're going to cover in the book, the functional reactive pixels repository has more examples of how to use view models with a photo details view controller and a login view controller. These demonstrate how to effectively use ReactiveCocoa to perform network operations and using `RACCommands` for responding to user interface events.

Testing View Models

The final lesson in this book is about testing. Specifically, *unit testing*. This is a controversial topic in the iOS community, which is partially why I've left it to last. Ideally, you'd be writing unit tests for your view models *as you write them*. However, learning how to use a new code paradigm is hard enough. Trying to test that which you don't yet understand can easily be overwhelming, so I've left it to the end.

I should also note that not everyone tests in the same way, or to the same degree. I come from a .Net background, where using mocks is common for testing the implementation details of a system under test. Others come from other backgrounds where mocking is less common, or even frowned upon. This section takes my approach to unit testing, but feel free to adopt it to your needs as you see fit.

Make sure your Podfile includes the following pods for the Tests target.

```
target "FRPTests" do

pod 'ReactiveCocoa', '2.1.4'
pod 'ReactiveViewModel', '0.1.1'
pod 'libextobjc', '0.3'
pod '500px-iOS-api', '1.0.5'
pod 'Specta', '~> 0.2.1'
pod 'Expecta', '~> 0.2'
pod 'OCMock', '~> 2.2.2'

end
```

And run `pod install`.

We're going to take a look at `FRPFullSizePhotoViewModel` first since it's the most vanilla Objective-C (it doesn't use much ReactiveCocoa). Here's its implementation again in its entirety.

```
@interface FRPFullSizePhotoViewModel ()

// Private access
@property (nonatomic, assign) NSInteger initialPhotoIndex;

@end

@implementation FRPFullSizePhotoViewModel

-(instancetype)initWithPhotoArray:(NSArray *)photoArray
    initialPhotoIndex:(NSInteger)initialPhotoIndex {
    self = [self initWithModel:photoArray];
    if (!self) return nil;

    self.initialPhotoIndex = initialPhotoIndex;

    return self;
}

-(NSString *)initialPhotoName {
    return [self.model[self.initialPhotoIndex] photoName];
}

-(FRPPhotoModel *)photoModelAtIndex:(NSInteger)index {
    if (index < 0 || index > self.model.count - 1) {
        // Index was out of bounds, return nil
        return nil;
    } else {
        return self.model[index];
    }
}

@end
```

Great. Let's test the initializer first, then move onto the other two methods.

We want to verify that when we initialize our view model, it has correctly assigned both the `model` and `initialPhotoIndex` properties.

```
#import <Specta/Specta.h>
#define EXP_SHORTHAND
#import <Expecta/Expecta.h>
#import <OCMock/OCMock.h>
#import "FRPPhotoModel.h"

#import "FRPFullSizePhotoViewModel.h"

SpecBegin(FRPFullSizePhotoViewModel)

describe(@"FRPFullSizePhotomodel", ^{
    it (@"should assign correct attributes when initialized", ^{
        NSArray *model = @[];
        NSInteger initialPhotoIndex = 1337;

        FRPFullSizePhotoViewModel *viewModel =
            [[FRPFullSizePhotoViewModel alloc]
                initWithPhotoArray:model
                initialPhotoIndex:initialPhotoIndex];

        expect(model).to.equal(viewModel.model);
        expect(initialPhotoIndex).to.equal(viewModel.initialPhotoIndex);
    });
});

SpecEnd
```

At the top, we have our `#import` statements, including a strange-looking `#define`. We keep that in there so we can use shorthand matchers like `expect()`. Then we have our private interface for the view model that we're testing. We need this to silence a compiler warning. Finally we have our test itself. The Specta test specification is fairly straightforward, and you can read more about it here, so I'm not going to go over it in detail in this book. In short, your test begins with `SpecBegin` and ends at `SpecEnd`. Tests go in the middle and have the format of `it(@"should ...", ^{ /* Write test code here */ });`.

Great! You can run the tests by stopping any current running app in the simulator and pressing Command-U. If everything worked, you should see the test pass.

Next, let's look at the `photoModelAtIndex:` method.

```
-(FRPPhotoModel *)photoModelAtIndex:(NSInteger)index {
    if (index < 0 || index > self.model.count - 1) {
        // Index was out of bounds, return nil
        return nil;
    } else {
        return self.model[index];
    }
}
```

It doesn't have *a lot* of logic in it, but as we'll see, other methods are going to be relying on it, so our tests should be robust.

```
it (@"should return nil for an out-of-bounds photo index", ^{
    NSArray *model = @[[NSObject new]];
    NSInteger initialPhotoIndex = 0;

    FRPFullSizePhotoViewModel *viewModel =
        [[FRPFullSizePhotoViewModel alloc]
            initWithPhotoArray:model
            initialPhotoIndex:initialPhotoIndex];

    id subzeroModel = [viewModel photoModelAtIndex:-1];
    expect(subzeroModel).to.beNil();

    id aboveBoundsModel = [viewModel photoModelAtIndex:model.count];
    expect(aboveBoundsModel).to.beNil();
});

it (@"should return the correct model for photoModelAtIndex:", ^{
    id photoModel = [NSObject new];
    NSArray *model = @[photoModel];
    NSInteger initialPhotoIndex = 0;

    FRPFullSizePhotoViewModel *viewModel = [[FRPFullSizePhotoViewModel alloc]
        initWithPhotoArray:model initialPhotoIndex:initialPhotoIndex];

    id returnedModel = [viewModel photoModelAtIndex:0];
    expect(returnedModel).to.equal(photoModel);
});
```

Awesome. Our two new tests have complete code coverage over the code under test. It checks for the three possibilities of the parameter for `photoModelAtIndex:`: less than zero, within bounds, and above bounds.

Finally, let's take a look at the `initialPhotoName` method.

```
-(NSString *)initialPhotoName {
    return [self.model[self.initialPhotoIndex] photoName];
}
```

This looks simple, but there's actually a lot going on here. To properly test this method, it's advisable to refactor some code and write a few different, smaller tests for that code.

```
-(NSString *)initialPhotoName {
    FRPPhotoModel *photoModel = [self initialPhotoModel];
    return [photoModel photoName];
}

-(FRPPhotoModel *)initialPhotoModel {
    return [self photoModelAtIndex:self.initialPhotoIndex];
}
```

This is a lot simpler. Each method does exactly one thing, and resembles the bark of a tree. Each layer depends on the inner layers. As long as we test everything, all the way down, we can be confident in our code.

The `initialPhotoModel` is a private method, so to test it we'll need to declare it in our test file.

```
@interface FRPFullSizePhotoViewModel ()

-(FRPPhotoModel *)initialPhotoModel;

@end
```

As you'll see, our test looks fairly simple.

```
it (@"should return the correct initial photo model", ^{
    NSArray *model = @[[NSObject new]];
    NSInteger initialPhotoIndex = 0;

    FRPFullSizePhotoViewModel *viewModel =
        [[FRPFullSizePhotoViewModel alloc]
            initWithPhotoArray:model
            initialPhotoIndex:initialPhotoIndex];
    id mockViewModel = [OCMockObject partialMockForObject:viewModel];

```

```
    [[[mockViewModel expect] andReturn:model[0]]
        photoModelAtIndex:initialPhotoIndex];

    id returnedObject = [mockViewModel initialPhotoModel];

    expect(returnedObject).to.equal(model[0]);

    [mockViewModel verify];
});
```

The test ensures that when `initialPhotoModel` is called, it then calls `photoModelAtindex:` with the `initialPhotoIndex`. The simplicity of the tests depends on us testing `photoModelAtIndex:` fully.

Next, let's take a look at the `FRPGalleryViewModel`. It's deceptively simple.

```
-(instancetype)init {
    self = [super init];
    if (!self) return nil;

    RAC(self, model) =
        [[[FRPPhotoImporter importPhotos] logError] catchTo:[RACSignal empty]];

    return self;
}
```

However, it's not very *testable*, and, in fact it needs to be refactored.

We'll refactor our view model to be a little simpler. Our new implementation looks like the following:

```
@implementation FRPGalleryViewModel

-(instancetype)init {
    self = [super init];
    if (!self) return nil;

    RAC(self, model) = [self importPhotosSignal];

    return self;
}

-(RACSignal *)importPhotosSignal {
    return [[[FRPPhotoImporter importPhotos] logError] catchTo:[RACSignal empty]];
}
```

```
15
16 @end
```

You can see that we've extracted out the `importPhotos` call in order to test whether or not that method gets called. We're not going to test `FRPPhotoImporter`, since it's outside the scope of this chapter.

Our new test looks like the following:

```
1  #import "Specta.h"
2  #import <OCMock/OCMock.h>
3
4  #import "FRPGalleryViewModel.h"
5
6  @interface FRPGalleryViewModel ()
7
8  -(RACSignal *)importPhotosSignal;
9
10 @end
11
12 SpecBegin(FRPGalleryViewModel)
13
14 describe(@"FRPGalleryViewModel", ^{
15     it(@"should be initialized and call importPhotos", ^{
16         id mockObject = [OCMockObject mockForClass:[FRPGalleryViewModel class]];
17         [[[mockObject expect] andReturn:[RACSignal empty]] importPhotosSignal];
18
19         mockObject = [mockObject init];
20
21         [mockObject verify];
22         [mockObject stopMocking];
23     });
24 });
```

That's a lot of code to test one method! I know, I know. It's a downfall of OCMock that it requires so much boilerplate, but you can't really blame it since it has to work within the confines of—*I shudder*—Objective-C.

We create a mock version of our gallery view model class, tell it to expect `importPhotoSignal` to be called, then initialize it. This gets a little tricky, since we're calling `init` on a mock object which *actually* subclasses `NSProxy`. However, OCMock is clever enough to figure it all out. It just looks really really weird. We're assigning `mockObject` to `[mockObject init]` to silence a compiler warning. Finally, we verify that all expected methods have been called.

The difficultly of testing illustrated in this example should serve as another reason why you should avoid side-effects in the initializer of a view model and prefer using `didBecomeActiveSignal`, instead.

Next we have our `FRPPhotoViewModel` class to test. Testing this class should highlight (again) the differences between causing side effects and using `didBecomeActiveSignal`. Let's take another quick look at the implementation:

```
@implementation FRPPhotoViewModel

-(instancetype)initWithModel:(FRPPhotoModel *)photoModel {
    self = [super initWithModel:photoModel];
    if (!self) return nil;

    @weakify(self);
    [self.didBecomeActiveSignal subscribeNext:^(id x) {
        @strongify(self);
        self.loading = YES;
        [[FRPPhotoImporter fetchPhotoDetails:self.model]
            subscribeError:^(NSError *error) {
            NSLog(@"Could not fetch photo details: %@", error);
        } completed:^{
            self.loading = NO;
            NSLog(@"Fetched photo details.");
        }];
    }];

    RAC(self, photoImage) =
        [RACObserve(self.model, fullsizedData) map:^id(id value) {
        return [UIImage imageWithData:value];
    }];

    return self;
}

-(NSString *)photoName {
    return self.model.photoName;
}

@end
```

Let's test the `photoName` method first, just to get it out of the way.

```
#import <Specta/Specta.h>
#define EXP_SHORTHAND
#import <Expecta/Expecta.h>
#import <OCMock/OCMock.h>

#import "FRPPhotoViewModel.h"
#import "FRPPhotoModel.h"

SpecBegin(FRPPhotoViewModel)

describe(@"FRPPhotoViewModel", ^{
    it (@"should return the photo's name property when photoName is invoked", ^{
        NSString *name = @"Ash";

        id mockPhotoModel = [OCMockObject mockForClass:[FRPPhotoModel class]];
        [[[mockPhotoModel stub] andReturn:name] photoName];

        FRPPhotoViewModel *viewModel = [[FRPPhotoViewModel alloc]
            initWithModel:nil];
        id mockViewModel = [OCMockObject partialMockForObject:viewModel];
        [[[mockViewModel stub] andReturn:mockPhotoModel] model];

        id returnedName = [mockViewModel photoName];

        expect(returnedName).to.equal(name);
        [mockPhotoModel stopMocking];
    });
});
```

We're inserting a mock photo model for the `model` property of our mock view model. It's really mocks all the way down.

Now onto the juicy initializer. This thing looks huge! Nearly twenty lines of unadulterated, untested code. Yikes! Let's simplify things a little bit and add our tests.

```
-(instancetype)initWithModel:(FRPPhotoModel *)photoModel {
    self = [super initWithModel:photoModel];
    if (!self) return nil;

    @weakify(self);
    [self.didBecomeActiveSignal subscribeNext:^(id x) {
        @strongify(self);
        [self downloadPhotoModelDetails];
    }];

    RAC(self, photoImage) =
        [RACObserve(self.model, fullsizedData) map:^id(id value) {
        return [UIImage imageWithData:value];
    }];

    return self;
}

-(void)downloadPhotoModelDetails {
    self.loading = YES;
    [[FRPPhotoImporter fetchPhotoDetails:self.model]
        subscribeError:^(NSError *error) {
        NSLog(@"Could not fetch photo details: %@", error);
    } completed:^{
        self.loading = NO;
        NSLog(@"Fetched photo details.");
    }];
}
```

We're choosing to not test `fetchPhotoDetails:` directly, so we put it in an instance method that we can easily test. The implementation details of this method aren't important to us. Let's do that now.

```
it (@"should download photo model details when it becomes active", ^{
    FRPPhotoViewModel *viewModel = [[FRPPhotoViewModel alloc] initWithModel:nil];

    id mockViewModel = [OCMockObject partialMockForObject:viewModel];
    [[mockViewModel expect] downloadPhotoModelDetails];

    [mockViewModel setActive:YES];

    [mockViewModel verify];
});
```

Notice how much more easy it is to test the view model when the side effects aren't in the initializer, but rather in a subscribe block for `didBecomeActiveSignal`.

No we need to test the rest of the view model. It's all fairly straightforward. We use fewer mocks, since a lot of our logic is just tied to how to map values from the view model's model to its own properties.

```
it (@"should return the photo's name property when photoName is invoked", ^{
    NSString *name = @"Ash";

    id mockPhotoModel = [OCMockObject mockForClass:[FRPPhotoModel class]];
    [[[mockPhotoModel stub] andReturn:name] photoName];

    FRPPhotoViewModel *viewModel = [[FRPPhotoViewModel alloc] initWithModel:nil];
    id mockViewModel = [OCMockObject partialMockForObject:viewModel];
    [[[mockViewModel stub] andReturn:mockPhotoModel] model];

    id returnedName = [mockViewModel photoName];

    expect(returnedName).to.equal(name);

    [mockPhotoModel stopMocking];
});

it (@"should correctly map image data to UIImage", ^{
    UIImage *image = [[UIImage alloc] init];
    NSData *imageData = [NSData data];

    id mockImage = [OCMockObject mockForClass:[UIImage class]];
    [[[mockImage stub] andReturn:image] imageWithData:imageData];

    FRPPhotoModel *photoModel = [[FRPPhotoModel alloc] init];
    photoModel.fullsizedData = imageData;

    __unused FRPPhotoViewModel *viewModel =
        [[FRPPhotoViewModel alloc] initWithModel:photoModel];

    [mockImage verify];
    [mockImage stopMocking];
});

it (@"should return the correct photo name", ^{
    NSString *name = @"Ash";
```

```
37
38      FRPPhotoModel *photoModel = [[FRPPhotoModel alloc] init];
39      photoModel.photoName = name;
40
41      FRPPhotoViewModel *viewModel =
42          [[FRPPhotoViewModel alloc] initWithModel:photoModel];
43
44      NSString *returnedName = [viewModel photoName];
45
46      expect(name).to.equal(returnedName);
47  });
```

That's it for writing unit tests for view models.

In an ideal world, writing tests should help change the way you write your code. Smaller, highly-cohesive methods are preferable to random side-effects all over the place. This simplicity perfectly exemplifies the ethos of functional reactive programming.

The nice thing about testing MVVM is that we're *not* touching UIKit. Remember that the hallmark of well-written MVVM view models is that the view models don't interact with the user interface classes *at all*.

Final Thoughts

MVVM is a really interesting paradigm; the more I think about it, the more it makes sense to me. Admittedly, the view models in this chapter are light on presentation logic. I'm going to beef them up on the GitHub repository, but this chapter should serve as an example of how to get started with MVVM.

I want to give a concrete example of using it where I feel that it makes more sense than MVC.

The example is the case of the view controller that's doing too much. On a recent app I helped build, we have a feed of items with pull to refresh. Tapping on an item navigates to that item's detail view controller. Standard stuff. However, sometimes the items within the feed were different. Sometimes they were results of the main feed from the API. Sometimes they were search results. Sometimes they were even static elements baked into the app at compile time.

Using MVC, there are two approaches that came to mind. First is the Massive View Controller where the one class knew how to display and manage all of the types of content. Not ideal. The other is subclassing an abstract base view controller that contains the logic common to all the types of content. This is the approach that I've taken in the past, but it makes refactoring difficult if, say, something that *was* common to all types of content becomes specific to only some types. It's also kind of a hack since Objective-C doesn't support abstract classes, anyway.

The approach I took was to use different view models that conformed to a protocol that the view controller could rely on. By placing the customized logic in the view models, I avoided the massive

view controller – it only knew how to display contents according to the view model protocol. Using MVVM is a great alternative to subclassing view controllers.

Additionally, if you have multiple platforms (say, iOS and OS X) they can use the *same view models* since they don't reference the view logic themselves. You could even go so far as to write generated view models in another language, and then generate view model objects in domain-specific languages like Objective-C, C#, Ruby, Java, or whatever language you need. Crazy stuff.

Finally, we didn't really get to touch on `RACCommand`. I'll lean on Justin Spahr-Summers a little bit to explain it in the context of MVVM.

1. A control is interacted with.
2. The command (belonging to the view model) is executed.
3. The view model's logic is run (this is the `signalBlock` that the command was initialized with).
4. The view model indirectly notifies the view through ReactiveCocoa, in our case, that the view should be updated.

Again, the GitHub repository contains more material on `RACCommand` that we're not covering in this book. Take a look.

MVVM works well, and it works well with ReactiveCocoa. You don't, however, need to go "whole hog" with it. You can start out small, using it with only one view controller to see how you like it. Try using it on your next project and you'll see how it can drastically simplify your view controller complexity.