Extracted from:

RubyMotion

iOS Development with Ruby

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

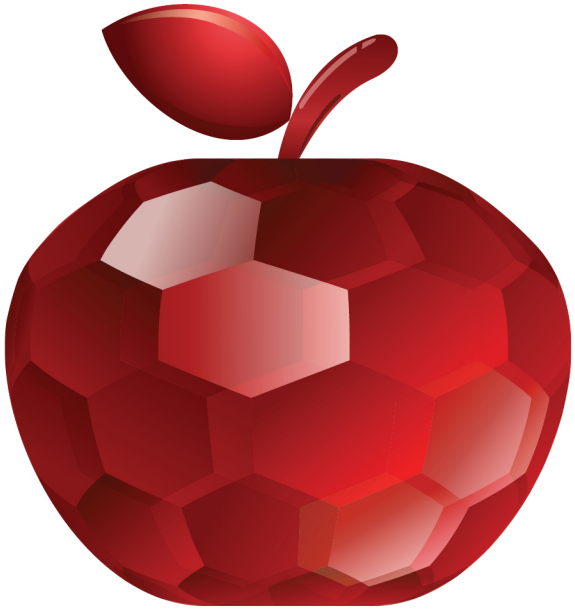
The Pragmatic Programmers

Pragmatic exPress

RubyMotion

iOS Development with Ruby

Updated for RubyMotion 2



Clay Allsopp

Foreword by Laurent Sansonetti,
lead developer of RubyMotion

edited by Fahmida Y. Rashid

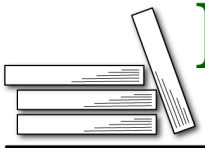
RubyMotion

iOS Development with Ruby

Clay Allsopp

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Fahmida Y. Rashid (editor)
Kim Wimpsett (copyeditor)
David J. Kelly (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-937785-28-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P3.0—July 2014

CHAPTER 3

Organizing Apps with Controllers

iOS apps usually consist of more than simple rectangles and buttons. We can easily build complex interfaces using the SDK; however, we need to first learn about *controllers* in order to create them.

Views are only one leg of the *Model-View-Controller* (MVC) programming paradigm adopted by the iOS SDK. A "programming paradigm" sounds intimidating, but MVC is actually fairly simple. The basic idea is that your code should have three types of objects: models to represent data, views to display those models, and controllers to process user input.

You can think of controllers as a layer between the user and the rest of your code. Their role is to interpret events and forward the changes to the relevant models and views. For example, tapping a button should be detected by a controller, which then increments a data property (model) and updates a label to reflect the change (view).

Controllers are instances of `UIViewController` in iOS. The SDK comes with several `UIViewController` subclasses with custom views and behavior to give every app the same look and feel. Controllers are absolutely central to iOS development, so we're going to take a look at how we use them.

Adding a New UIViewController

As the name suggests, `UIViewControllers` are objects that control a view. The `UIViewController` object stores the `UIView` it manages inside the `view` attribute. However, we generally don't use `addSubview:` to add this particular view to the screen; instead, various methods will often take the entire `UIViewController` object and adjust the view as necessary before adding it to a hierarchy.

Let's get started with a small project to see how UIViewController works, and you'll see what I mean. We'll create an app showing different ways to explore colors, specifically, motion create ColorViewer.

First we make the ./app/controllers directory (mkdir ./app/controllers). This is where we'll keep all of our controller classes. When building production-level apps, you should also add views and models subdirectories, but we won't be needing those right now.

Then we add a colors_controller.rb file in controllers. This will be our custom UIViewController subclass that's presented to the user. We'll start by setting its superclass and adding one short method.

controllers/ColorViewer/app/controllers/colors_controller.rb

```
class ColorsController < UIViewController
  def viewDidLoad
    super

    self.view.backgroundColor = UIColor.whiteColor

    @label = UILabel.alloc.initWithFrame(CGRectZero)
    @label.text = "Colors"
    @label.sizeToFit
    @label.center =
      [self.view.frame.size.width / 2,
       self.view.frame.size.height / 2]
    @label.autoresizingMask =
      UIViewAutoresizingFlexibleBottomMargin | UIViewAutoresizingFlexibleTopMargin
    self.view.addSubview(@label)
  end
end
```

Subclassing UIViewController always involves overriding viewDidLoad(). This method is called right after our controller's view has been created and is where we do whatever custom setup is necessary. For now, we just set the view's background color, add a label, and call it a day.

viewDidLoad() is one of the *view life-cycle* methods. Every controller's view goes through several stages: creation, appearance, disappearance, and destruction. You can add custom behaviors at each point using the corresponding life-cycle methods, but the most common is viewDidLoad().

Now that we have our controller, open our AppDelegate. We're going to create a UIWindow just like we did in the previous chapter, except we're now going to use the rootViewController=() method instead of addSubview:.

controllers/ColorViewer/app/app_delegate.rb

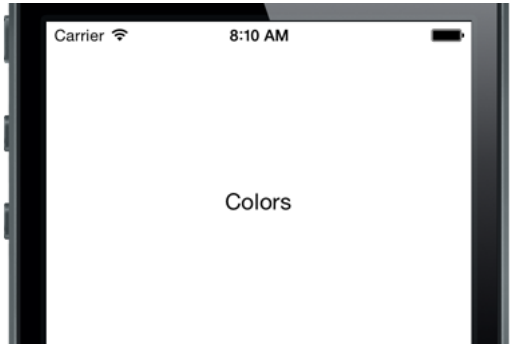
```
class AppDelegate
  def application(application, didFinishLaunchingWithOptions:launchOptions)
    @window = UIWindow.alloc.initWithFrame(UIScreen.mainScreen.bounds)
    @window.makeKeyAndVisible

    @window.rootViewController =
      ColorsController.alloc.initWithNibName(nil, bundle: nil)

    true
  end
end
```

rootViewController=() will take the UIViewController and adjust the view's frame to fit the window. This lets us write our controller without hard-coding its size, making our controller reusable to other containers. As we said earlier, methods in which you pass a UIViewController are very common, as we'll soon see.

We instantiate UIViewControllers with initWithNibName:bundle:. This method can be used to load a controller from a .NIB/.XIB file created using Xcode's Interface Builder, but in this case, we passed nil, meaning the controller will be created programatically.[1]

initWithNibName:bundle: is the *designated initializer* of UIViewController. Whenever you want to create a controller, you *must* call this method at some point, either directly or inside the definitions of your custom initializers (such as controller.initWithSome:property:).

Let's run our app and check it out:



1. RubyMotion does support Interface Builder; simply add your .NIB or .XIB files to the project's ./resources directory. Using Interface Builder is beyond the scope of this text, but you can use the IB RubyGem (https://github.com/RubyMotion/ib) to help connect your Ruby- Motion code inside Interface Builder.

As we saw in *Making Shapes and Colors, on page ?*, a few addSubview:s could have given us the same result, but using controllers creates a view that can easily fit into different containers. In fact, we're going to do just that with UINavigationController in *Using Multiple Controllers with UINavigationController, on page 8*.

super and the Life-Cycle Methods

It's a good habit to call super() in viewDidLoad() and the other life-cycle methods. The default implementations of UIViewController can have important setup details, and you may experience some unexpected and hard-to-debug behavior if you don't call them; this is particularly true if you start subclassing UINavigationController or UITableViewController.

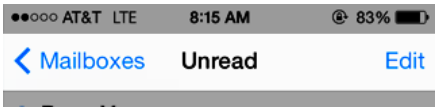
In some cases, notably viewDidAppear, Apple explicitly says you need to call the superclass method in any UIViewController subclass. Refer to Apple's documentation[a] for more details.

a. http://developer.apple.com/library/ios/#documentation/uikit/reference/UIViewController_Class/Reference/Reference.html

Using Multiple Controllers with UINavigationController

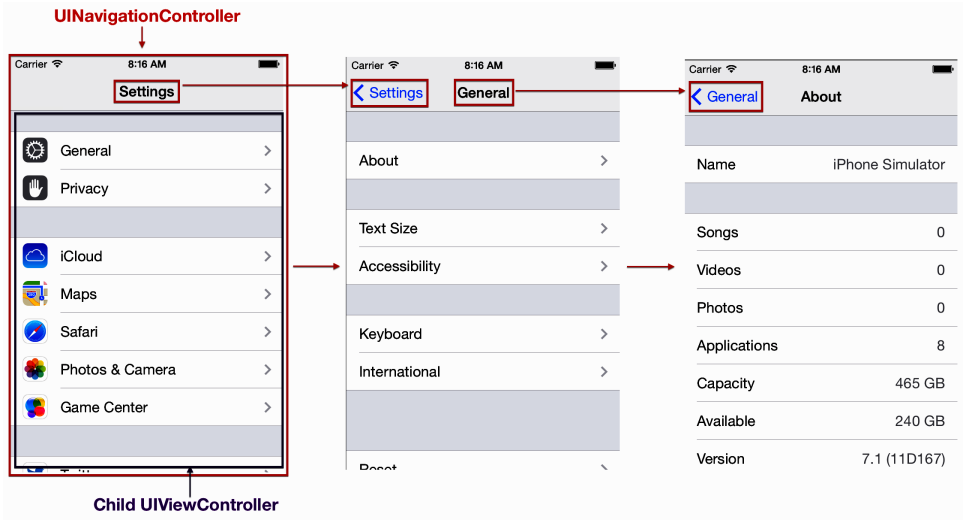
Although some iOS apps are famous for their unique visuals, most apps share a common set of interface elements and interactions included with the SDK. Typical apps will have a persistent top bar (usually blue) with a title and some buttons; these apps use an instance of UINavigationController, one of the standard *container controllers* in iOS.

Containers are UIViewController subclasses that manage many other *child* UIViewControllers. Kind of wild, right? Containers have a view just like normal controllers, to which their children controllers' views are added as subviews. Containers add their own UI around their children and resize their subviews accordingly. UINavigationController adds a navigation bar and fits the children controllers below, like this:



UINavigationController manages its children in a stack, pushing and popping views on and off the screen. Visually, new views are pushed in from the right, while old views are popped to the left. For example, Mail.app uses this to dig down from an inbox to an individual message. UINavigationController also automatically handles adding the back button and title for you; all you need to worry about

is pushing and popping the controller objects you're using. Check out the following to see how Settings.app uses navigation controllers.



UINavigationController is pretty easy to integrate. In AppDelegate, we change our rootViewController() assignment to use a new UINavigationController.

controllers/ColorViewer_nav/app/app_delegate.rb

```
class AppDelegate
  def application(application, didFinishLaunchingWithOptions:launchOptions)
    @window = UIWindow.alloc.initWithFrame(UIScreen.mainScreen.bounds)
    @window.makeKeyAndVisible
➤   controller = ColorsController.alloc.initWithNibName(nil, bundle: nil)
➤   nav_controller =
➤     UINavigationController.alloc.initWithRootViewController(controller)
➤   @window.rootViewController = nav_controller
    true
  end
end
```

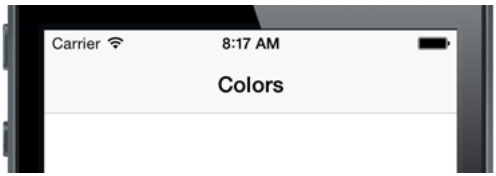
initWithRootViewController: will take the given controller and start the navigation stack with it. As we said earlier, the UINavigationController will handle adding and resizing this controller's view to fit to the appropriate size.

Before we run the app, we should make one more change in ColorsController. Every UIViewController has a title(), which UINavigationController uses to set the top bar's title.

controllers/ColorViewer_nav/app/controllers/colors_controller.rb

```
self.view.addSubview(@label)
➤ self.title = "Colors"
```

Run and check out our slightly prettier app:



Excellent, let's make it do something. We'll add a few buttons to our view, each representing a color. When a button is tapped, we'll push a detail controller with its color as the background. Visually, a new controller will slide in from the right while the old ColorsController fades to the left.

First we need to add those buttons to the view at the end of viewDidLoad(). We're going to use some of Ruby's dynamic features to get this done, primarily the send() method. You can use any of the default UIColor helper methods like purpleColor() or yellowColor(), but we're going to stick with the basics.

controllers/ColorViewer_nav/app/controllers/colors_controller.rb

```
["red", "green", "blue"].each_with_index do |color_text, index|
  color = UIColor.send("#{color_text}Color")
  button_width = 80

  button = UIButton.buttonWithType(UIButtonTypeSystem)
  button.setTitle(color_text, forState:UIControlStateNormal)
  button.setTitleColor(color, forState:UIControlStateNormal)
  button.sizeToFit
  button.frame = [
    [30 + index*(button_width + 10),
     @label.frame.origin.y + button.frame.size.height + 30],
    [80, button.frame.size.height]
  ]
  button.autoresizingMask =
    UIViewAutoresizingFlexibleBottomMargin | UIViewAutoresizingFlexibleTopMargin
  button.addTarget(self,
    action:"tap_#{color_text}",
    forControlEvents:UIControlEventTouchUpInside)
  self.view.addSubview(button)
end
```

See the color = UIColor.send("#{color_text}Color") trick? This lets us create the UIColor, button text, and button callback all with a single color_text variable. If you run our app now, you should see the three buttons just like this (but don't tap any quite yet):

Now on to implementing those button callbacks. For each of these, we'll need to push a new view controller onto our UINavigationController's stack. UIViewControllers happens to have a navigationController() property, which lets us access the parent UINavigationController. This navigationController() is automatically set whenever we add a view controller to a navigation stack, which we did with initWithRootViewController:. With that in mind, our button callbacks look something like this:

controllers/ColorViewer_nav/app/controllers/colors_controller.rb

```
def tap_red
  controller = ColorDetailController.alloc.initWithColor(UIColor.redColor)
  self.navigationController.pushViewController(controller, animated: true)
end
def tap_green
  controller = ColorDetailController.alloc.initWithColor(UIColor.greenColor)
  self.navigationController.pushViewController(controller, animated: true)
end
def tap_blue
  controller = ColorDetailController.alloc.initWithColor(UIColor.blueColor)
  self.navigationController.pushViewController(controller, animated: true)
end
```

We call pushViewController:animated: on the navigation controller, which pushes the passed controller onto the stack. By default, the navigation controller will also create a back button that will handle popping the frontmost child controller for us. If you need to do that programmatically, just call popViewControllerAnimated:() on UINavigationController.

We referenced a new ColorDetailController class, so let's implement that. First we create color_detail_controller.rb in ./app/controllers. We'll keep it simple and just set the title and background color.

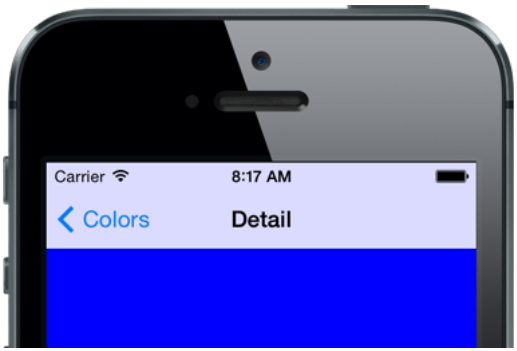
controllers/ColorViewer_nav/app/controllers/color_detail_controller.rb

```
class ColorDetailController < UIViewController
  def initWithColor(color)
    self.initWithNibName(nil, bundle:nil)
    @color = color
    self.title = "Detail"
    self
  end
  def viewDidLoad
    super

    self.view.backgroundColor = @color
  end
end
```

Start by defining a new initializer, initWithColor:, which takes a UIColor as its argument. Our implementation of this method uses UIViewController's designated

initializer initWithNibName:bundle:(), which is required for any controller initializer we write. You also need to return self from these functions, which should make sense given all the times we assign a variable from these methods (such as controller = UIViewController.alloc.initWithNibName(nil, bundle: nil)).

It's time to rake and play with our navigation stack. It should look like this:

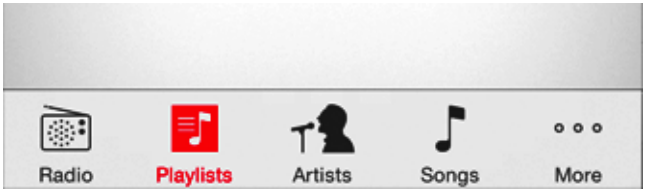


Check out the slick animations on the navigation bar, where the title simultaneously fades and slides as a new controller is pushed.

Many apps structure their interface using UINavigationController, where each pushed controller gradually reveals more detailed data. As you saw, there are only a couple of methods we need to implement that user interface. However, some apps need more than this kind of hierarchal layout. UITabBarController is another widely used container controller, and in *Separating Controllers with UITabBarController*, on page 12 we're going to add it to our app.

Separating Controllers with UITabBarController

UITabBarController functions an awful lot like UINavigationController. The children controllers' views fit above the black tab bar, where each tab corresponds to one child. The Music app shows a tab with room for four controllers:



The fact that the More tab appears here indicates there are more than five children.

Unlike other containers, UITabBarControllers are *only* to be used as the rootViewController() of a UIWindow. You cannot push an instance of UITabBarController in

pushViewController:animated:. From a user-experience perspective, this means you should use a tab bar only if contains very distinct and globally applicable controllers.

Just like UINavigationController, tab bars are easy to add. It just takes a small change to AppDelegate.

controllers/ColorViewer_tab/app/app_delegate.rb

```
controller = ColorsController.alloc.initWithNibName(nil, bundle: nil)
nav_controller =
  UINavigationController.alloc.initWithRootViewController(controller)
tab_controller =
  UITabBarController.alloc.initWithNibName(nil, bundle: nil)
tab_controller.viewControllers = [nav_controller]
@window.rootViewController = tab_controller
```

We create a UITabBarController like a normal UIViewController and set its viewControllers() to an array containing our navigation controller. The order of viewControllers() corresponds to the left-to-right order of the tabs.

When you run our app, you can see the top and bottom bars typical to most iOS apps implemented with a pretty small amount of code. Since it's not very helpful to have just one unstyled tab, let's fix that.

Every UIViewController has a tabBarItem() property, which accepts UITabBarItem, an object containing information about how to draw the view for the controller in the bottom tab bar. It is *not* a UIView but rather a plain object that the system uses to construct a view. We use the UITabBarItem to customize the icon, title, and other appearance options for the controller's tab.

The first step is to override initWithNibName:bundle: in ColorsController, and then we can create our UITabBarItem.

controllers/ColorViewer_tab/app/controllers/colors_controller.rb

```
def initWithNibName(name, bundle: bundle)
  super
  self.tabBarItem =
    UITabBarItem.alloc.initWithTitle(
      "Colors",
      image: nil,
      tag: 1)
  self
end
```

initWithTitle:image:tag: is one initializer for UITabBarItem, which we can use to set a custom image and title. tag: can be used to uniquely identify the tab bar item, but we won't be using it here. image should be a 30x30 black and transparent icon. Setting image to nil means we won't display images here.

You can also use the initWithTabBarSystemItem:tag: initializer to automatically set the title and image, assuming your tab corresponds to one of the default styles (such as Favorites or Contacts).

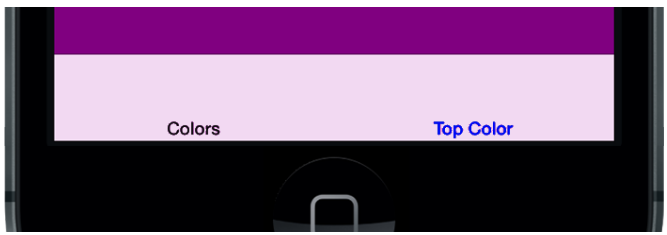
Why did we create our tab item in initWithNibName:bundle:? We want to create the tabBarItem() as soon as the controller exists, regardless of whether its view has been created yet. UITabBarController will load the views only when absolutely necessary, so if you wait to create the tab item in viewDidLoad(), then some controllers' items might not be set when the app is done launching.

One more thing! We should probably add another tab, right? We'll pretend this is the Top Color section, where we can view the most popular color. This way we can reuse our ColorDetailController.

controllers/ColorViewer_tab/app/app_delegate.rb

```
top_controller = ColorDetailController.alloc.initWithColor(UIColor.purpleColor)
top_controller.title = "Top Color"
top_nav_controller =
  UINavigationController.alloc.initWithRootViewController(top_controller)
tab_controller.viewControllers = [nav_controller, top_nav_controller]
```

Run the rake command once again, and *voila*! You should see a whole bunch of container controllers like this:



Before we add even more content to our app, let's take a moment to examine a few of the more subtle details of controllers.

The Edges of UIViewControllers

If you take a look at our second tab, you should notice that the purple background extends *underneath* the tab bar. In fact, all of our colored screens extend underneath the default iOS navigation elements. How does that happen?

Prior to iOS7, the navigation bar, tab bar, and other interface elements were opaque, so any interior controllers were sized to always be visible on the screen. But with the release of iOS7, many interface elements switched to a translucent visual effect, and Apple encourages apps to take advantage of that by layering their controllers and views.

This present a few problems: what if we don't want important items to be hidden below the bars? What if we don't want anything underneath? Apple provides a few methods on UIViewController to help us out.

If you don't want any part of your controller to show up underneath the navigation chrome, you can use the edgesForExtendedLayout() property of your UIViewController. For example, to prevent our second tab from leaking its color under the tab bar, we just change this property to UIRectEdgeNone:

controllers/ColorViewer_edge/app/app_delegate.rb

```
top_controller.title = "Top Color"
top_controller.edgesForExtendedLayout = UIRectEdgeNone
top_nav_controller =
  UINavigationController.alloc.initWithRootViewController(top_controller)
tab_controller.viewControllers = [nav_controller, top_nav_controller]
```

Try out the app and see how the purple is no longer slipping under the navigation bar or tab bar. We could even set the edgesForExtendedLayout() property somewhere in the definition of our UIViewController subclass, like viewDidLoad(), if we never wanted this class to extend its edges.

Our controllers don't do a whole lot, but you can see how these two classes form the building blocks of many iOS apps. UINavigationController and UITabBarController provide easy ways to organize many different parts of your app, but what if we really need to focus the user's attention on just one screen? Well, it turns out that we can also present controllers *modally* in front of all other controllers.

Presenting Modal UIViewControllers

Sometimes we want one controller to take up the entire screen to get a user's attention. For example, Mail.app's New Message screen appears on top of the usual inbox list, forcing the user to either complete the message or explicitly end the task. To accomplish this, we present the controller modally.

UIViewControllers allows us to present modal view controllers at any point in their life cycle. The key method is presentViewController:animated:completion:, which functions similarly to UINavigationController's pushViewController:animated:. The given controller will be presented above all other controllers in the app and will remain there until we invoke dismissViewControllerAnimated:completion:.

Let's present a modal controller from our Top Color controller. The presented controller will allow us to change the top color, which is definitely a task best done while the rest of the interface is obscured.

First we need a button in the navigation bar at the top of the screen that presents this modal controller. We saw UIButton in Chapter 2, *Filling the Screen with Views*, on page ?, but we need to use a different class for bar buttons: UIBarButtonItem. This isn't a subclass of UIView; instead, it's a plain-old Ruby object that we use to specify the text, image, and style of the bar button. The system will then take care of how to draw and add the button specifications as a view, much like UITabBarItem.

Let's add our bar button to our app. In ColorDetailController, we create the button item in viewDidLoad() like so:

controllers/ColorViewer_modal/app/controllers/color_detail_controller.rb

```
  rightButton =
    UIBarButtonItem.alloc.initWithTitle("Change",
      style: UIBarButtonItemStyleBordered,
      target:self,
      action:'change_color')
  self.navigationItem.rightBarButtonItem = rightButton
end
```

We create our UIBarButtonItem instance with a title and a style. The style property determines how our button looks: it can be plain, bordered, or "done" (play around to see the difference). We then set the new UIBarButtonItem as our controller's navigationItem()'s rightBarButtonItem(). Every UIViewController has a navigationItem(), which is how we access all the information displayed in the top bar. Again, note that UINavigationItem is *not* a UIView, so you cannot add new subviews to it.

We also assign a target and action in the initializer, which function in the same manner as when we call addTarget:action:forControlEvents: on a UIButton.

We haven't implemented change_color() yet, so let's get to it. To make modal controllers stand out even more, it's a common practice to wrap them in a small UINavigationController. All we need to do is call presentViewController:animated:completion: with that controller, so it's short and sweet.

controllers/ColorViewer_modal/app/controllers/color_detail_controller.rb

```
def change_color
  controller = ChangeColorController.alloc.initWithNibName(nil, bundle:nil)
  controller.color_detail_controller = self
  self.presentViewController(
    UINavigationController.alloc.initWithRootViewController(controller),
    animated:true,
    completion: lambda {})
end
```

Everything looks normal except the lambda in completion. Just like the view animations we saw in *Animating Views*, on page ?, presenting controllers

take an anonymous callback function. We don't need to do any special behavior right now, but it's there if you ever need it. Our presented controller is a new ChangeColorController object, which is a class we don't have yet.

Create change_color_controller.rb in ./app/controllers; this will be the controller that we actually present. It won't be a super-complicated class: we'll add a text field for the user to enter the color, as well as a button to enact that change. But before all that, we need to set up the plumbing.

controllers/ColorViewer_modal/app/controllers/change_color_controller.rb

```
class ChangeColorController < UIViewController
  attr_accessor :color_detail_controller
```

We start with Ruby's nifty attr_accessor() to create the methods color_detail_controller() and color_detail_controller=(). We need these so we can easily store a reference to the ColorDetailController whose color we're changing.

The ChangeColorController modal view also needs a UITextField and UIButton, both of which we covered in Chapter 2, *Filling the Screen with Views*, on page ?. When the button is tapped, we'll take whatever is in the text field and use that to create a UIColor that ColorDetailController can use. Just like the other controllers, this logic belongs in viewDidLoad().

controllers/ColorViewer_modal/app/controllers/change_color_controller.rb

```
def viewDidLoad
  super
  self.title = "Change Color"
  self.view.backgroundColor = UIColor.whiteColor
  @text_field = UITextField.alloc.initWithFrame(CGRectZero)
  @text_field.borderStyle = UITextBorderStyleRoundedRect
  @text_field.textAlignment = UITextAlignmentCenter
  @text_field.placeholder = "Enter a color"
  @text_field.frame = [CGPointZero, [150,32]]
  @text_field.center =
    [self.view.frame.size.width / 2, self.view.frame.size.height / 2 - 170]
  self.view.addSubview(@text_field)
  @button = UIButton.buttonWithType(UIButtonTypeSystem)
  @button.setTitle("Change", forState:UIControlStateNormal)
  @button.frame = [[
     @text_field.frame.origin.x,
     @text_field.frame.origin.y + @text_field.frame.size.height + 10
    ],
    @text_field.frame.size]
  self.view.addSubview(@button)
  @button.addTarget(self,
    action:"change_color",
    forControlEvents:UIControlEventTouchUpInside)
end
```

It's lengthy, but most of the code just lays out our views. We position @text_field slightly above the center of the view and then position @button right below. Finally, we set our callback to be change_color(), so we need to write that too.

controllers/ColorViewer_modal/app/controllers/change_color_controller.rb

```
  def change_color
    color_text = @text_field.text
    color_text ||= ""
    color_text = color_text.downcase
    color_method = "#{color_text}Color"
    if UIColor.respond_to?(color_method)
      color = UIColor.send(color_method)
      self.color_detail_controller.view.backgroundColor = color
      self.dismissViewControllerAnimated(true, completion: nil)
      return
    end

    @text_field.text = "Error!"
  end
end
```

First we try to generate a UIColor from @text_field.text. We use respond_to?() to catch invalid colors (like "catdogColor"), but if nothing bad happens, then we forge ahead. We grab a reference to our ColorDetailController using color_detail_controller(), set its background color, and then dismiss ourselves with dismissViewControllerAnimated:completion:(). Not bad at all, right?

Take our app for a spin, and everything should go smoothly, as in the following figure:

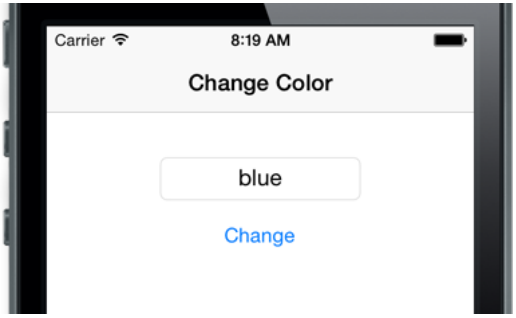


Figure 6—Changing the color from a text field

Make sure to test our exception handling in ChangeColorController, as well some less-known UIColor helpers like magenta or cyan.

We made some really tangible progress in this chapter as we built software that looks and acts like what's expected on iOS. The code we went through should give you a better idea about the biggest difference between the iOS APIs and plain Ruby: the original Objective-C method names are very verbose. Need I say even more than UIViewAutoresizingFlexibleBottomMargin? Thankfully, we pulled a few Ruby tricks with UIColor.send and saved even more boilerplate by removing the need for header files and complex class definitions just with attr_accessor().

Our apps can now be organized using the standard UI patterns, but that last example showed a significant gap in our knowledge: changing data can be messy. Passing the ColorDetailController as a property and altering its view directly is less than desirable. We're going to cover a more automatic way of handling those kind of data changes and more in Chapter 4, *Representing Data with Models*, on page ?.