Showcase App

The focus of this practice is to delve deeper into Xcode's graphical interface design tool, Interface Builder. Interface Builder has always been a key part of the Xcode set of development tools. However, with the release of Xcode 4, Interface Builder became part of Xcode itself, as opposed to previous versions in which it was a separate application. What makes Interface Builder an attractive addition to Apple's developer tools is that it removes the need to write code in order to design great interfaces for your applications. It allows you to lay out your views and windows by dragging built-in Cocoa objects from the Object Library and placing them on the screen.

What's even more useful is that by using the Attributes Inspector, you can make many changes that would otherwise require lines upon lines of code. As a developer, this is good news for two reasons. First, you don't have to continuously test, build, and run your application in order to see if what you're designing with code looks good. With Interface Builder, you can see this right away. Second, similar to what's just been mentioned, you can make changes graphically, which saves a lot of time and effort. All this—plus using Interface Builder makes designing views fun!

This practice explains how to set up an application using the Tabbed Application template (Fig 1).

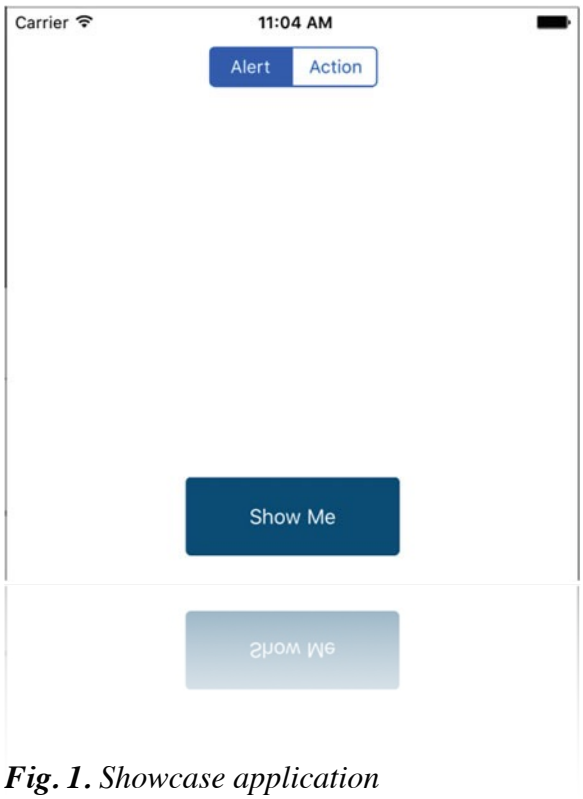

Fig. 1. *Showcase application*

The great thing about using a Tabbed Application is that each of the tabs can act as an app within an app, each one showing a drastically different set of tools and styles. The two initial tabs you set up will showcase some of the interface elements you haven't seen yet, as well as use your device's

GPS function. Once you've done this, you set up a third tab that will allow you to demonstrate some of the important interface elements that can't be added using Interface Builder. A goal of this practice is to show how much you can achieve while using as little code as possible, and it's important to note how little code this example requires compared to how much you would need to write if Interface Builder were not a part of Xcode. The last thing you look at is how you can alter interface elements with code to achieve results that Interface Builder alone can't do but that are important in building beautiful, easy-to-use interfaces. Here is an outline of what each of the tabs will include:

- *Track It*: Here you create a text view that displays detailed telemetry from the GPS receiver, on either a physical device or in the simulator. You also use a switch to turn the GPS on and off.

- *Slide It*: As the name implies, in the second tab you look at how to implement the slider tab, where to build a series of sliders to alter the background color of the entire view, and how to output their values into text fields. You also learn the answer to one of the burning questions all iOS developers ask: "How do I dismiss the keyboard?"

- *Alert*: In the final tab, you see how to use a segmented control to determine what happens when a button is pushed. The choice is between an alert view and an action sheet, two popular elements in many applications.

Getting Ready

Let's get to it:

1. Open Xcode and create a new project by clicking Create A New Xcode Project on the Welcome screen or choosing File ➤ New ➤ Project (+Shift+N). Select the Tabbed Application template and click Next.

2. Name your project Showcase and ensure that the device is set to iPhone. Configure the other settings.

3. You don't want to create a Git repository, so leave the Source Control option unchecked. Ensure that your project is going to be saved where you want it to be and click Create.

You've now created the bare bones of your Showcase tabbed application. To see what Apple's template has given you as a starting point, click Main.Storyboard; you should see a screen resembling that shown in Fig. 2.

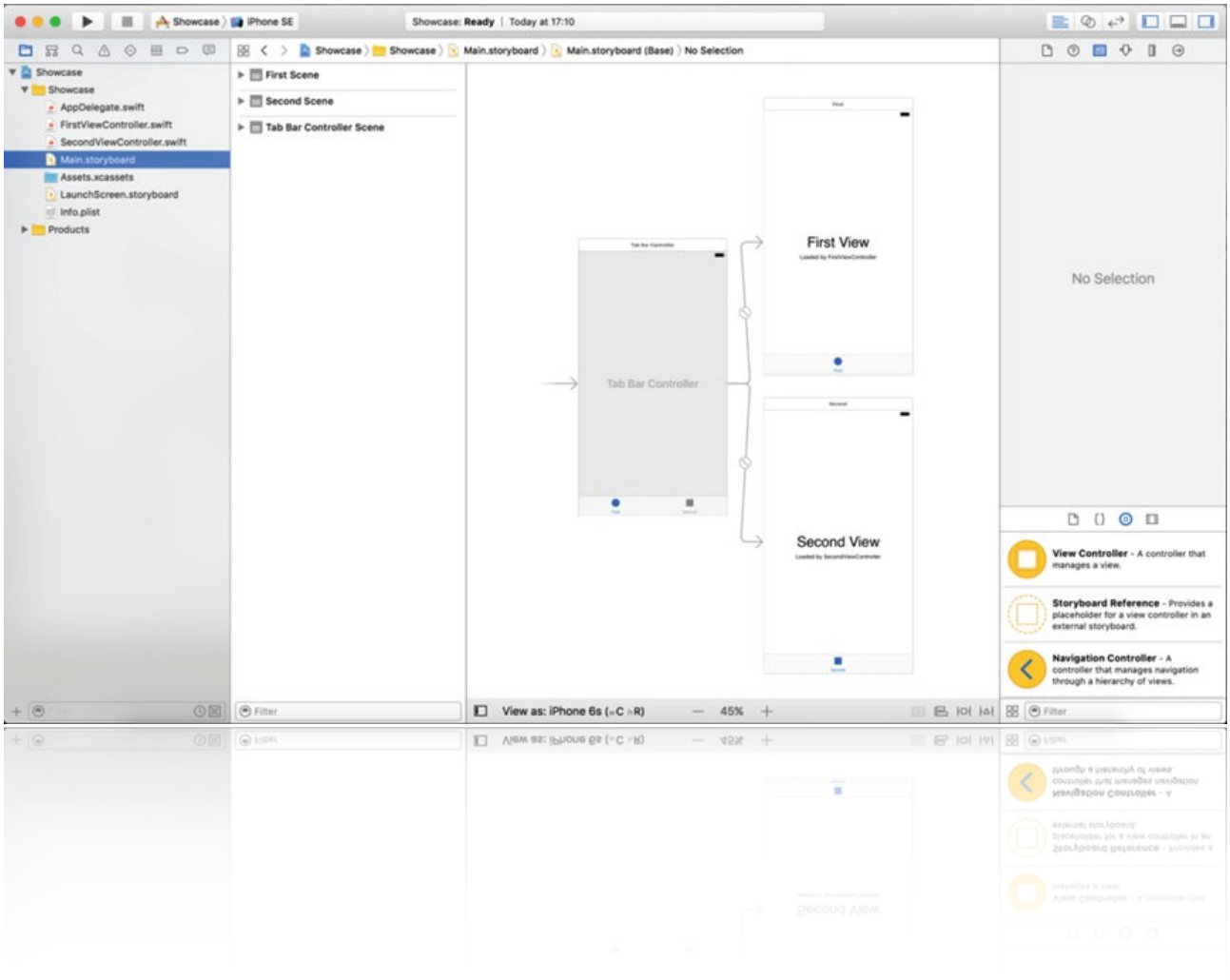

Fig. 2. *The starting point for the Showcase application*

By default, the template gives you a tab bar controller (UITabBarController) with two view controllers (UIViewController) attached named FirstViewController and SecondViewController. Although these names are perfectly good, tab orders can change as a project develops, so it's always better to use names that are semantically accurate. So, before you add a third tab, let's rename the files to something more appropriate.

With the Project Navigator open (+1), highlight the FirstViewController.swift file and press Return on your keyboard. You should now be able to rename the file. Remove the text and type TrackViewController.swift (remember to add the .swift extension). Repeat this for SecondViewController.swift, but call it SliderViewController.swift. Your Project Navigator should closely resemble what you see in Fig. 3.

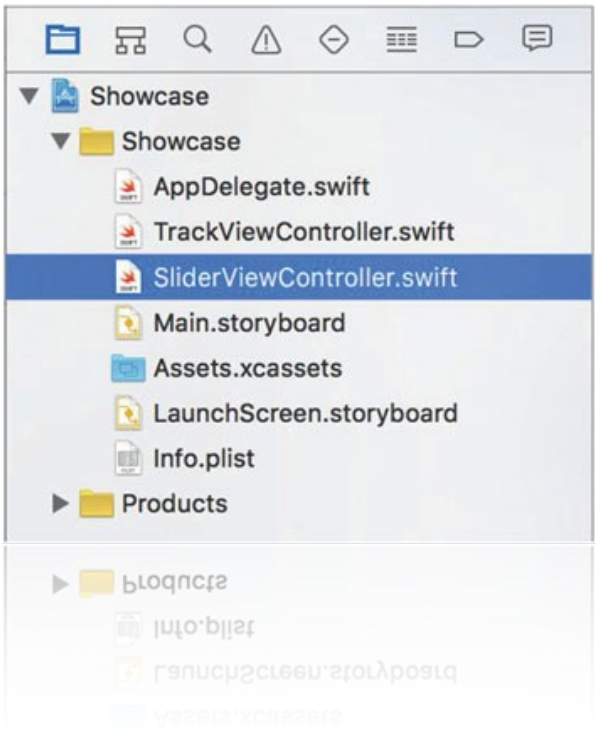

Fig. 3. *The current project's files as shown in the ProjectNavigator*

Next you need to update your code files to use these new filenames, and to do this you use the Find Navigator (+3). You need to set up the Find Navigator to rename every instance of FirstViewController to TrackViewController. By default, you see Find ➤ Text ➤ Containing above the search criteria. Click the word Find, and select Replace ➤ Text ➤ Containing, as shown in Fig. 4.

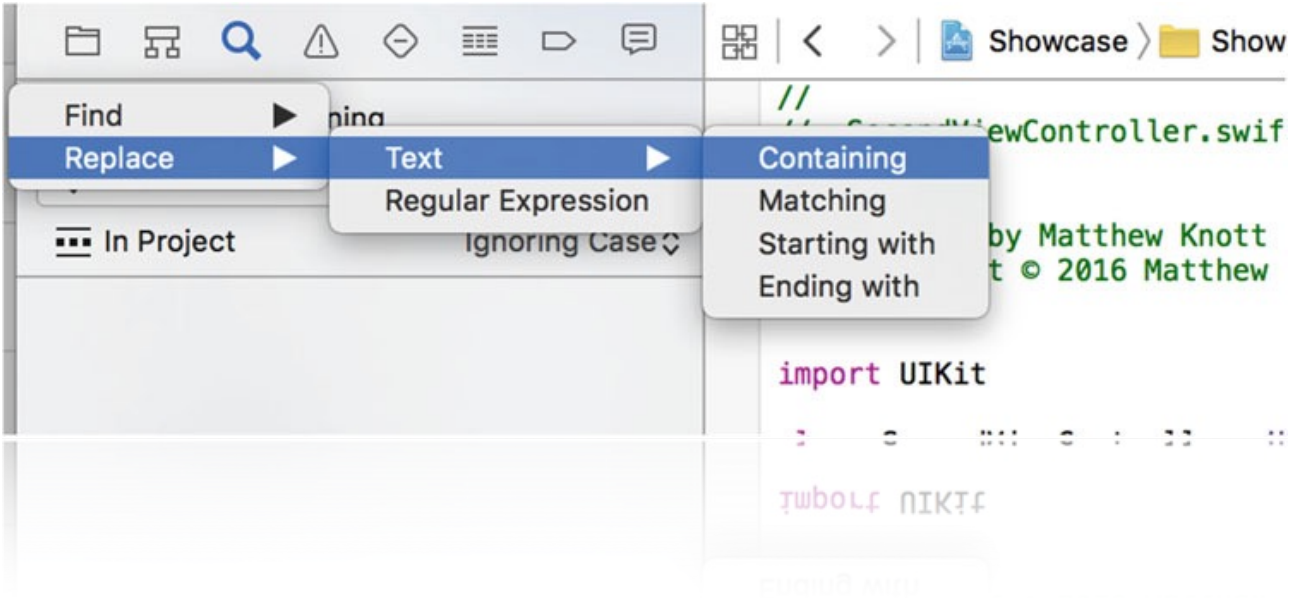

Fig. 4. *Configuring the Find Navigator to perform a find-and-replace task*

In the first text field, type FirstViewController, and in the second, type TrackViewController. At this point you encounter an uncharacteristically poor piece of interface design: you need to press Return to perform the search, although Xcode doesn't make this clear. Click Replace All, and Xcode will go through all the files listed and replace the word FirstViewController with TrackViewController. Fig. 5 illustrates the results of this find-and-replace operation in the Find Navigator.

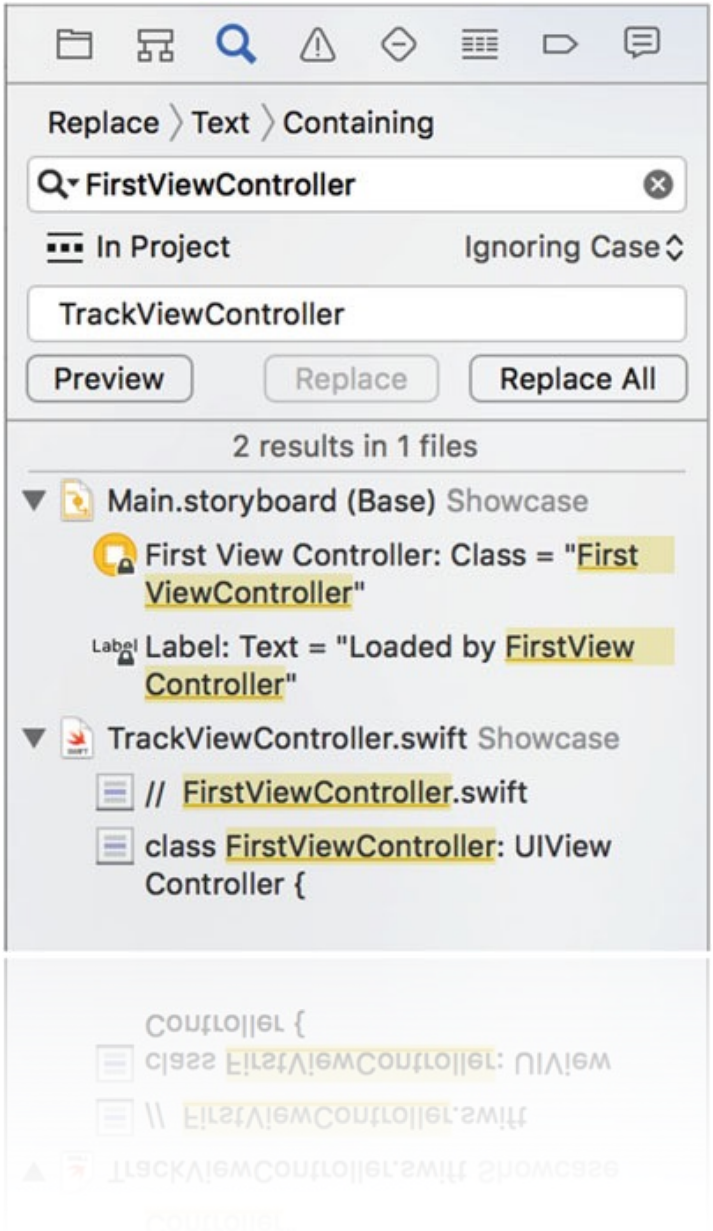

Fig. 5. *The Find Navigator updating references to old file and classnames*

Once the find-and-replace operation has completed, repeat the task, but in the first box enter SecondViewController as the text you're searching for and in the second type SliderViewController as the text you want to replace it with. Press Return and then click Replace All.

You've now updated all references to your renamed view controllers. Next you create a third view controller called ActionViewController:

1. Switch back to the Project Navigator from the Find Navigator so you're ready to start interacting with the project files again.

2. Create a new file (+N). Select Source from the left sidebar under iOS, and then choose Cocoa Touch Class, as you did during the previous practice. Click Next.

3. Specify ActionViewController as the class name. Type UIViewController in the Subclass Of field, and ensure that Also Create XIB File is *not* checked. Click Next. Create this file in the same location where all your other files are stored and click Create.

Adding Tab Bar Icons to an Asset Catalog

Since they were introduced in Xcode 5, Asset Catalogs have been used to store the icons that appear on tabs in the Tabbed Application template. Although this isn't the way you *have* to store the images, it's certainly best practice, and it makes storing retina and regular versions of the same icon much easier and less cluttered.

For this project, we have created three purpose-built icons. If you don't want to create your own icons, a range of free tab- bar icons created by Charlene are available for download at www.iconbeast.com. Once you've downloaded the icons, you're ready to begin working with the Asset Catalog in this project:

1. In Xcode, select Assets.xcassets. You should see the three images shown in Fig. 6.

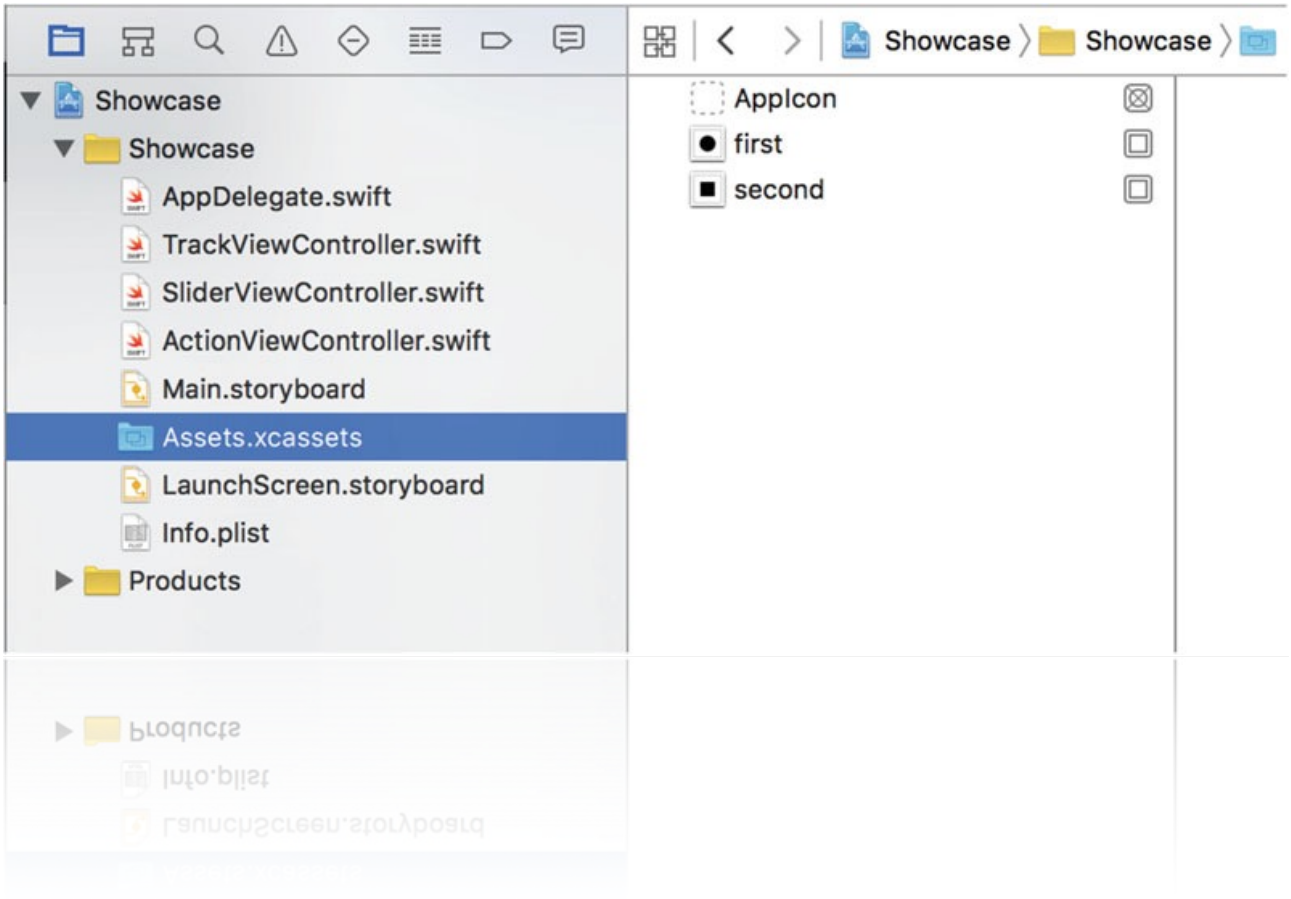

Fig. 6. *The contents of the Assets.xcassetslibrary*

2. Select the image named first and delete it by pressing the Backspace key or right-clicking and selecting Remove Selected Items. Repeat this step for the image named second.

3. Click the plus symbol at the bottom of the list of images, and from the menu that appears, select New Image Set, as shown in Fig. 7. This creates a new image set called image.

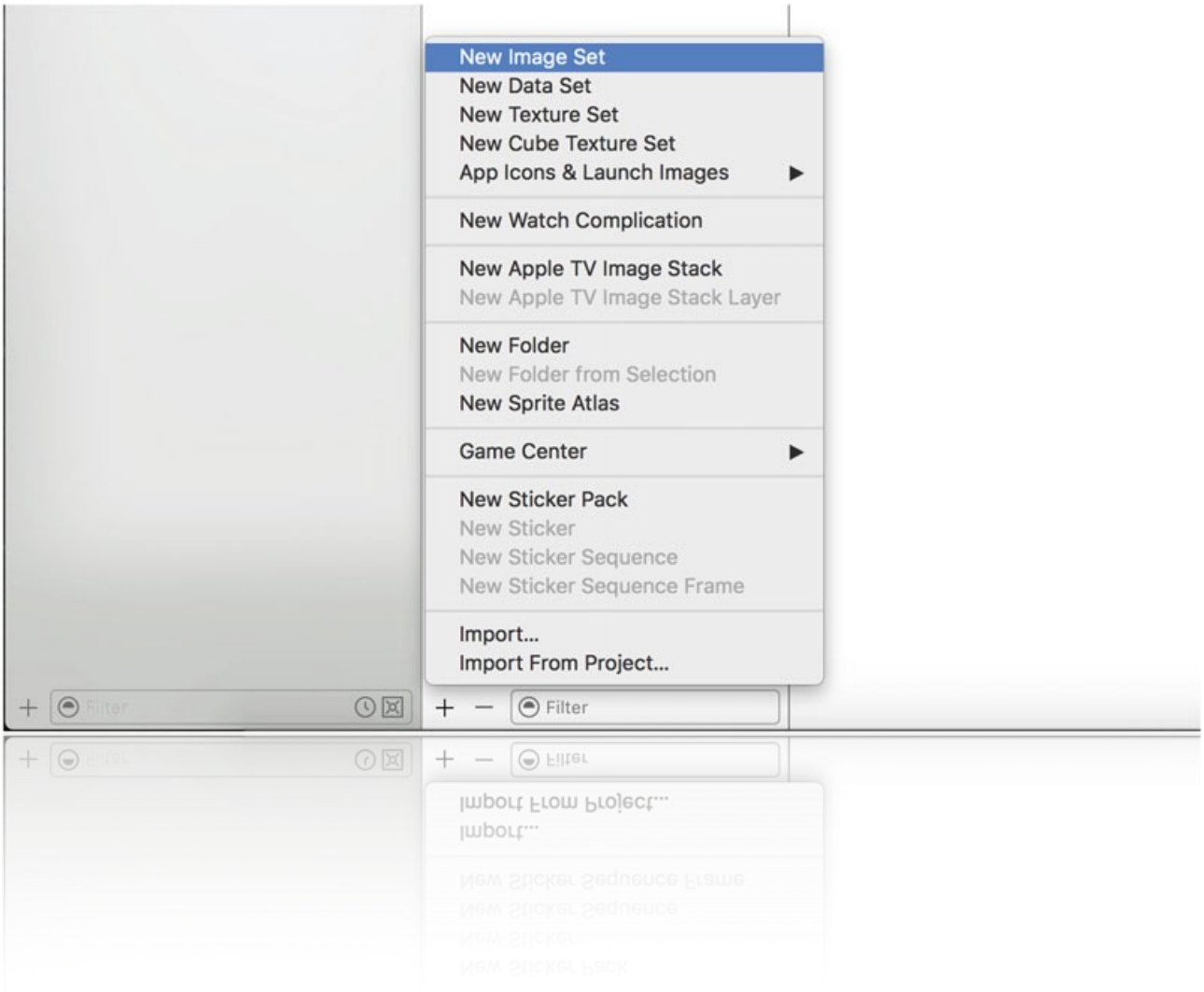

Fig. 7. *Creating a new image set*

4. Select the new image set and press Return so that you can rename the file; rename it Track. Repeat Step 3 twice to create two more new image sets, naming them Slider and Alert, respectively. You've now created three image sets, which will contain the tab-bar icons for the three tabs.

5. Open a Finder window and browse to where you extracted the icons.

6. Select the Track image set in Xcode. You see something resembling the screen in Fig. 8.

Fig. 8. *The three new image sets*

One of the benefits of using Asset Catalogs for storing images is that they make it easy to group identical images that have multiple resolutions, which you need in order to ensure your app displays properly on any compatible device. Within the image set, there are three containers: a 1x container, 2x container, and a
3x container. Traditionally, in the 1x container you would place the standard- resolution image, and in the 2x container the retina, or higher-resolution image. Because of the even higher resolutions of devices such as the iPhone 6 plus and retina iPads, Apple has introduced a 3x container to ensure your icons are crisp and sharp. This image set does not rigidly enforce a specific icon resolution, but as a guideline for tab-bar icons, use 30px × 30px for standard-resolution icons, 60px × 60px for 2x retina icons, and 90px × 90px for 3x retina icons, which is the exact resolutions of the icons you downloaded.

7. In the Finder window, locate the icon named mapicon.png, and drag it into the 1x container. Then drag the mapicon@2x.png file into the 2x container and mapicon@3x.png into the 3x container.

8. Repeat Step 7 for the two remaining image sets, dragging slidericon.png, slidericon@2x.png, and slidericon@3x.png into the containers for the Slider image set, and alerticon.png, alerticon@2x.png, and alerticon@3x.png into the Alert image set. Your Asset Catalog should now resemble that shown in Fig. 9.

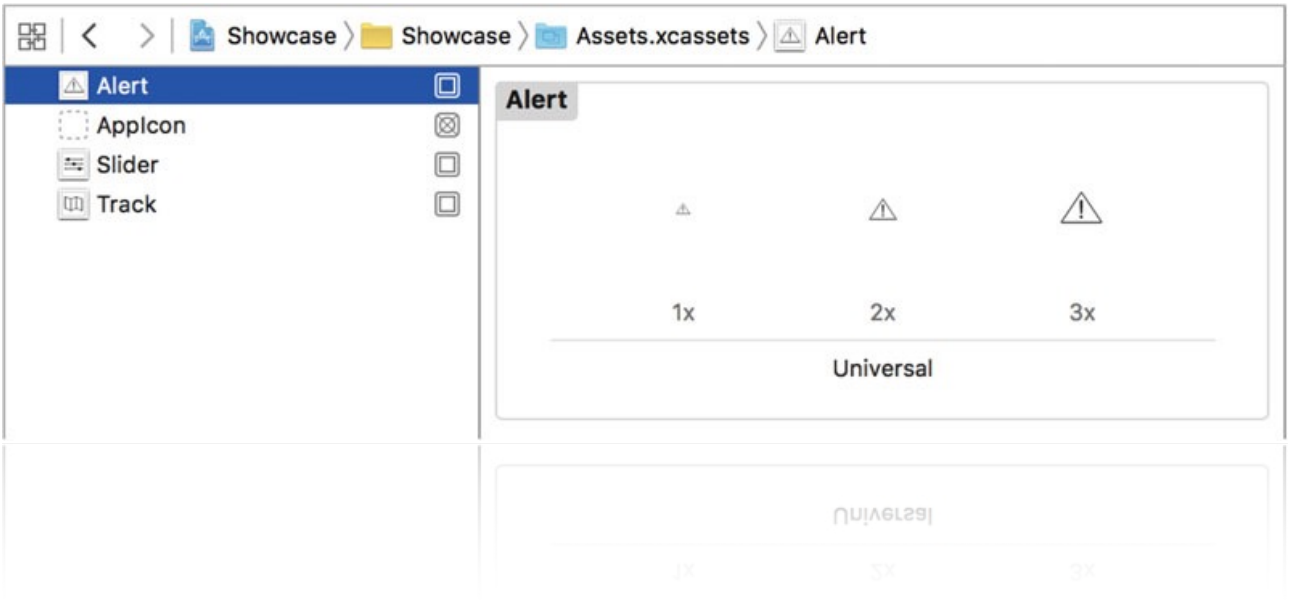

Fig. 9. *The three image sets with the icons in place*

The benefit to using Asset Catalogs for managing images is that you're left with a much neater project in the Project Navigator. Now you're set up and can begin working on your interfaces using Interface Builder. Let's start by taking a closer look at the different areas of the Interface Builder.

Before You Start

Storyboards are a relatively new feature of Xcode. They allow you to logically lay out how views are connected, pushed, and managed as a user navigates through your application. They can greatly simplify applications, plus they add a degree of logic to how you develop your projects.

Because you work with them a little more in this practice than the last, it's important to know the basics because they are now a part of Interface Builder. Open Main.storyboard, and let's take a look at the key controls. Conveniently, all the controls for your storyboard are located at the bottom of the storyboard design area and are separated into three groups, as shown in Fig. 10.

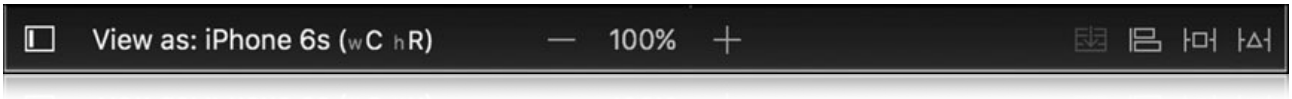

Fig. 10. *The storyboard controls*

Let's look at the groups and their icons:

- *Document Outline toggle*: This first button, located by itself in the bottom-left corner of the design area, hides and displays the document outline.

- *Form Factor toggle*: Located in the middle of the design area's icons, this control allows you to alter how view controllers in the storyboard are displayed. This is incredibly useful if you're designing a single interface for multiple form factors, because you can quickly move

between orientations and sizes and have that reflected for the entire storyboard.

- *Zoom controls*: Apple finally added a quick zoom control to interface builder in Xcode 8. As expected, this allows you to control the zoom of the design area.

- *Constraint controls*: These four buttons, which are grouped together, let you control the behavior of the elements in your view when faced with differing resolutions or different screen resolutions:

    - *Stack*: Allows you to group selected elements into a stack view.

    - *Align*: Allows you to position elements in relation to the view, letting you set a range of alignments including centering and aligning to the top of the view.

    - *Pin*: Fixes an element in place by manually setting its constraints.

    - *Resolve Auto Layout Issues*: One of the most useful buttons in Xcode. You can often use the powerful auto-layout functions offered from this menu to do all the hard work for you.
    There are no buttons to control zoom level; Apple requires developers to handle zoom by using the pinch and squeeze gestures on a multitouch-enabled device, using a scroll wheel, or by double-clicking the whitespace around the views to snap in or out of the storyboard.
    This concludes our brief look at the storyboard design area controls. Let's move on and build the interface.

Building the Interface

Now that you're familiar with the storyboard controls, you can start to get your interface in order. To do that, you first remove the two views that were added by default. With Main.storyboard selected, you have two view controllers, as you saw in Fig. 2. Just like you did during previous practice, you begin by removing the bulk of the boilerplate content so that you can see for yourself exactly how the different elements are created:

1. Above each of the two view controllers is a bar. Click the bar, and three icons appear as shown in Fig. 11. Press the Backspace key to delete the bar.

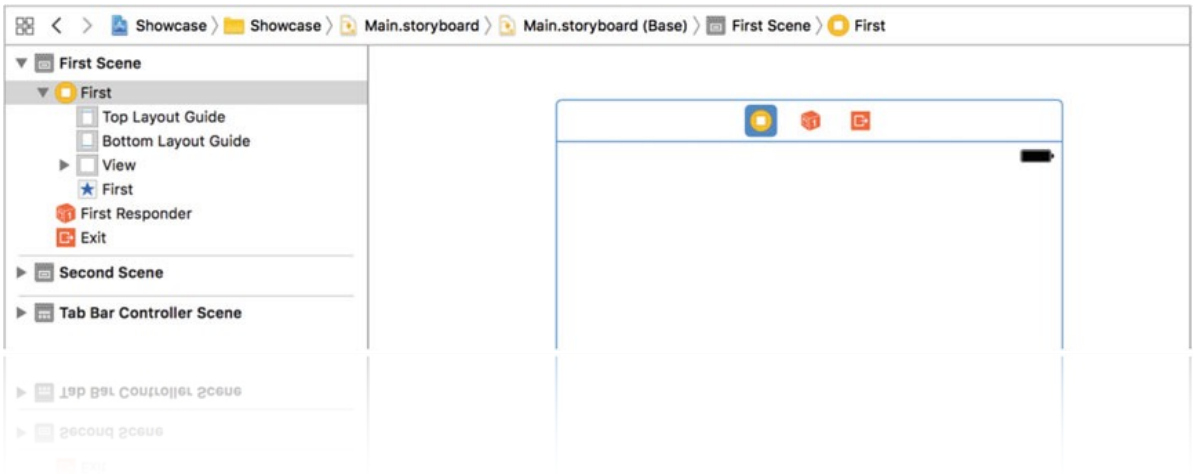

Fig. 11. *The viewcontroller, outlined in blue after selecting the topbar*

2. Repeat this step for the other view controller so you're left with only a tab bar controller.

3. All your views are going to be based on standard view controllers, so locate the view controller object in the Object Library and drag three of them to the design area. Position them as shown in Fig. 12.

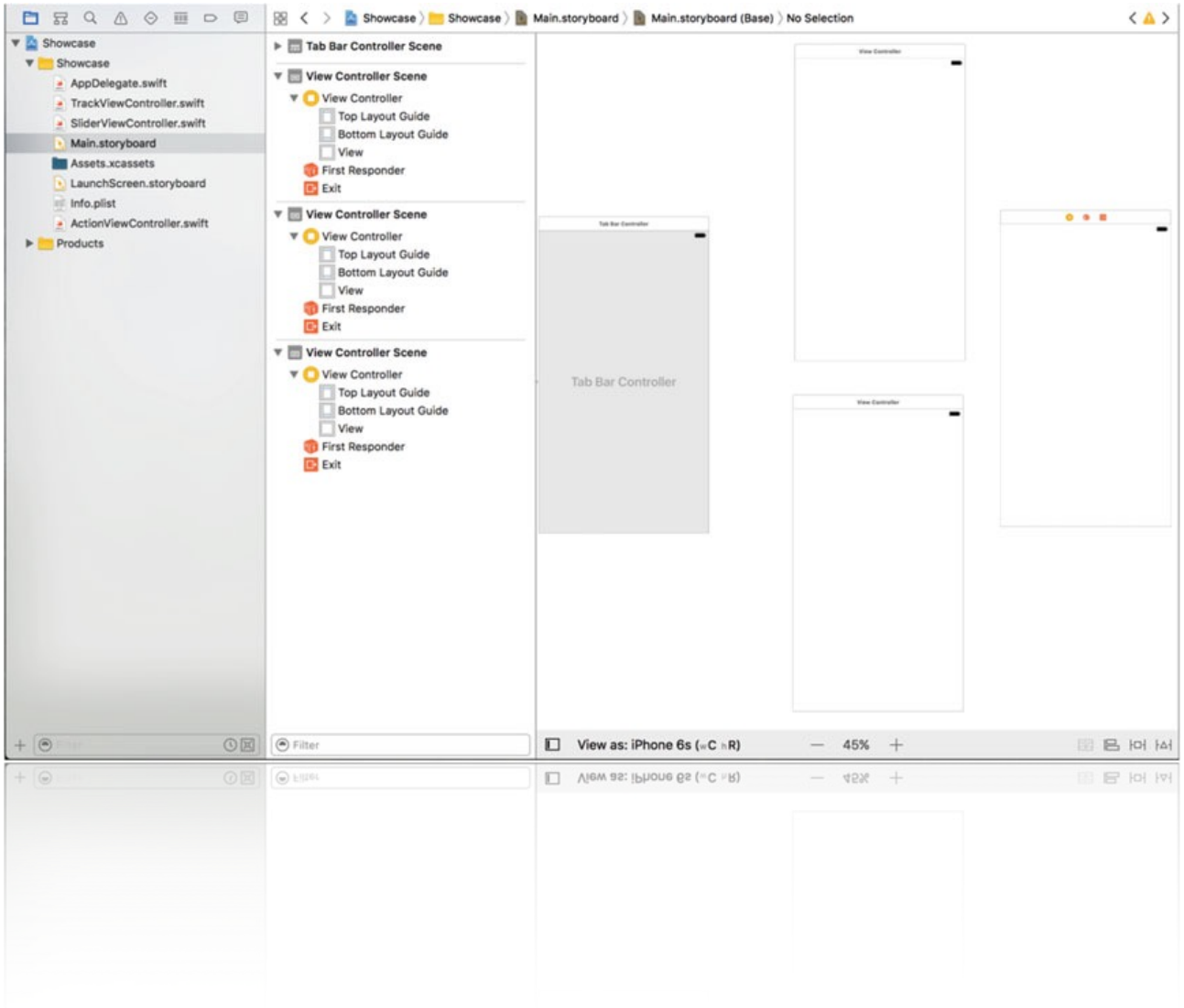

Fig. 12. *Main.storyboard with the three orphaned view controllers. The Utilities pane can be hidden to give extra room.*

Although you've added the three view controllers, they're currently orphans—that is, there is no relationship between the view controllers and the tab bar controller, so in order for the application to run and display a view, you need to create one. The process for creating a relationship between the tab bar controller and the view controllers is similar to how you connected objects to their actions and methods during the previous practice:

1. Set the zoom level so that you can see all of the view controllers.

2. Select the tab bar controller by clicking it once.

3. Holding the control key (^), click the tab bar controller and drag a connection to the top view controller, as shown in Fig. 13.

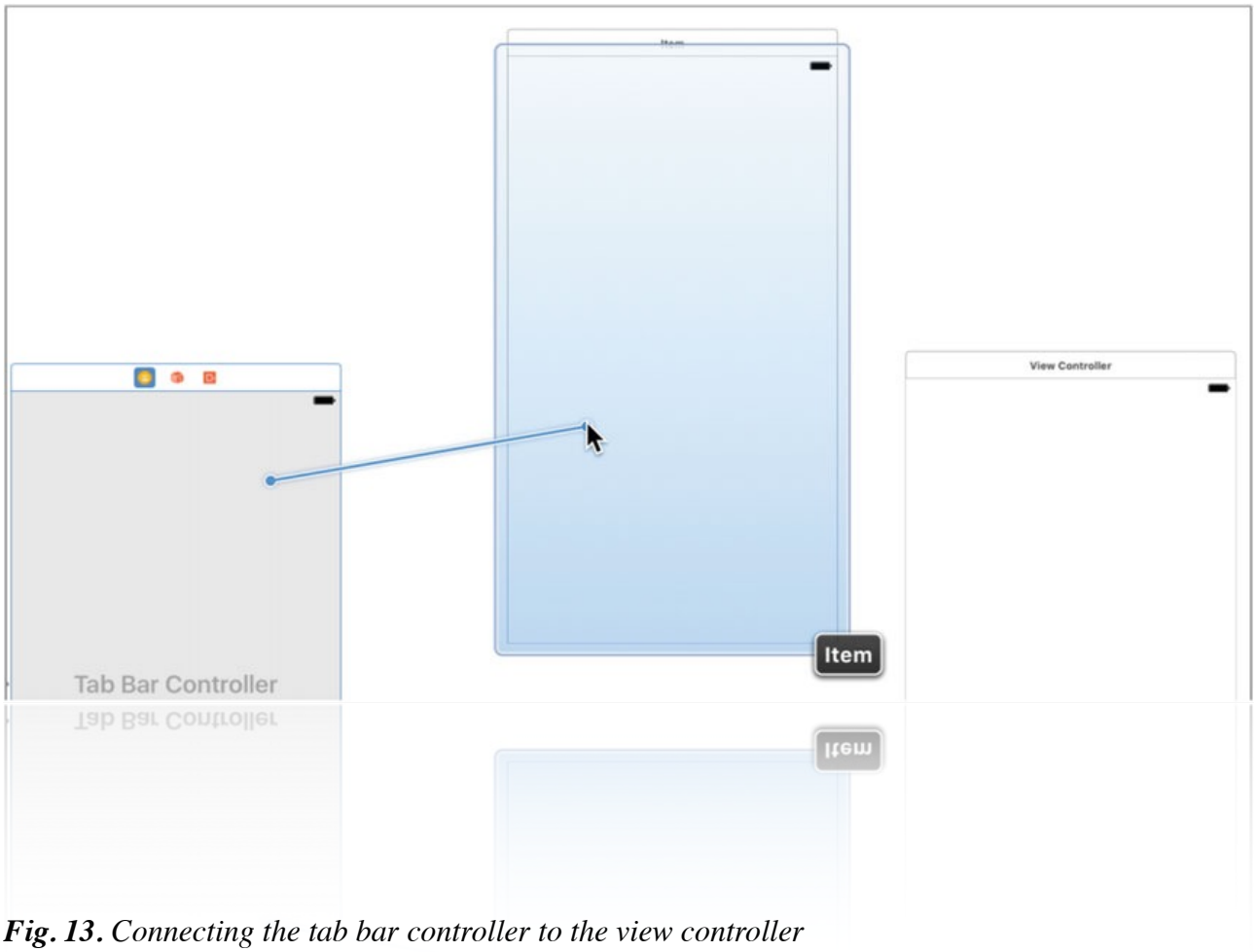

Fig. 13. *Connecting the tab bar controller to the view controller*

4. When you release the mouse button, a dialog appears, asking you to choose the segue type. Select View Controllers under the Relationship Segue heading, as shown in Fig. 14.

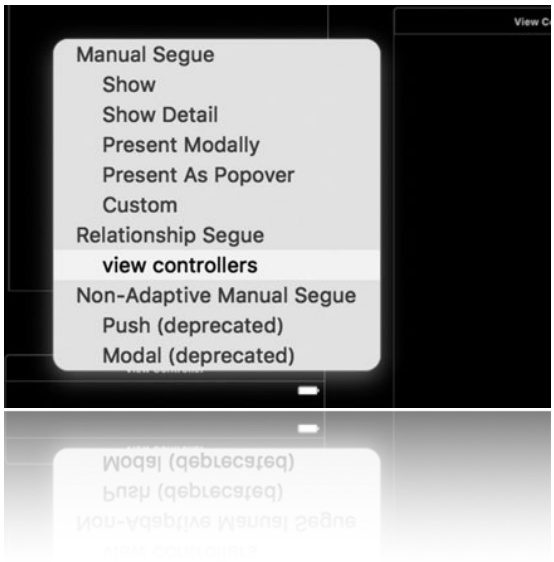

Fig. 14. *The segue selection dialog*

Now that you've created a relationship between the tab bar controller and the view controller, notice that a few changes have been made in your design area. The tab bar controller has a tab showing on the tab bar, as does the view controller. Also, a line connects the tab bar controller to the top view controller; this is called a segue, and it's a visual representation of the relationship between two elements in a storyboard. Segues can link elements in several different ways, but on this occasion you only choose the View Controllers branch to create a relationship segue.

5. Repeat Step 4 for the remaining two view controllers—first the bottom view controller, then the middle one—until you're left with something resembling the screen shown in Fig. 15.

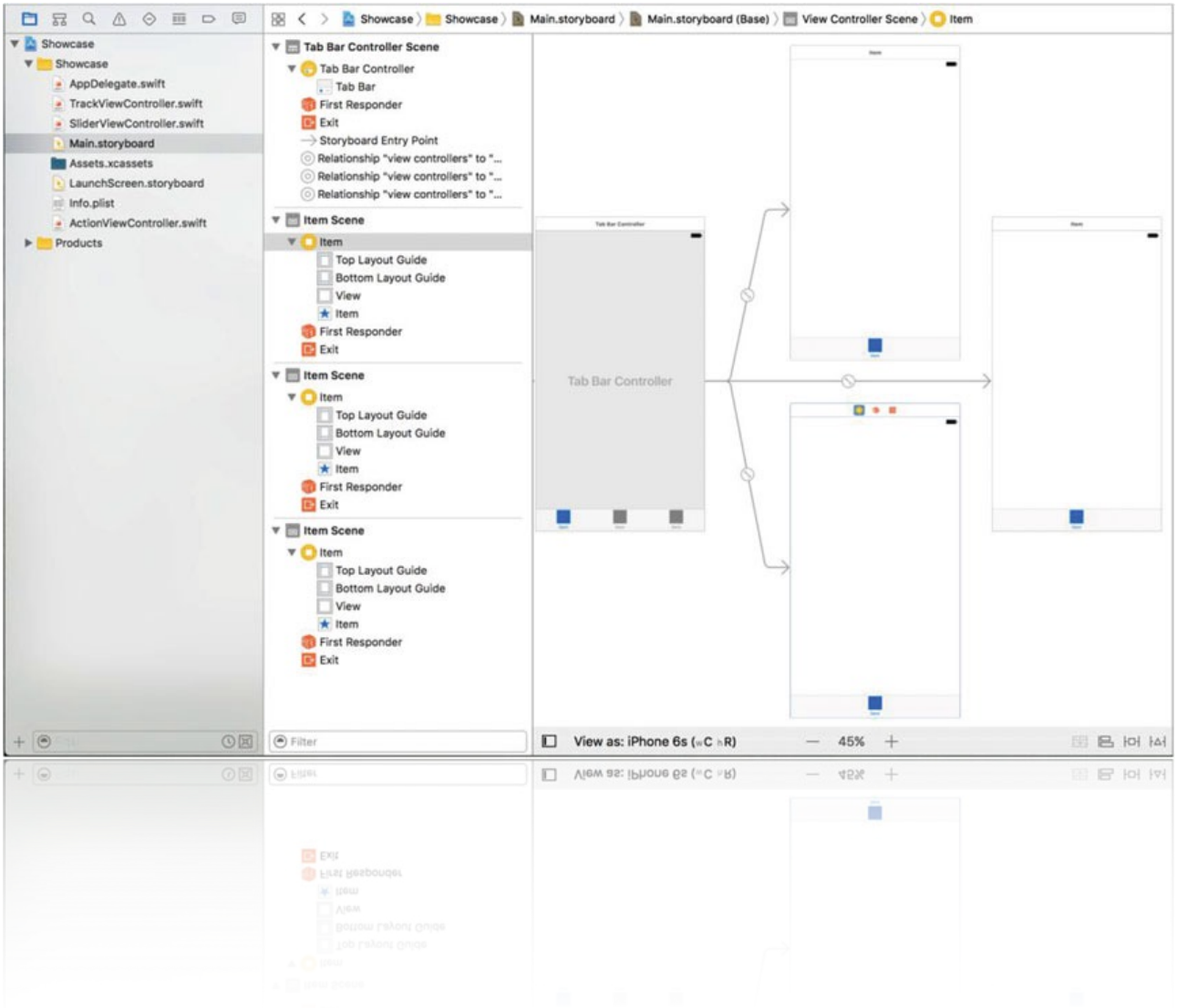

Fig. 15. *The view controllers are all connected to the tab bar controller*

Setting the Tab Icons

You're nearly ready to focus on the individual views in your application, but before you do, there are a couple more tasks to complete. You need to implement the icons you added to the image Asset Catalog:

1. Zoom back in to the design area.

2. Locate the topmost view controller and select the square icon above the text Item, as shown in Fig. 16.

Fig. 16. *Selecting the tabbar icon in the topview controller*

3. Open the Attributes Inspector (+ +4). Set the Title attribute to Track It and the Image attribute to Track. Be sure not to set the Selected Image attribute—it needs to stay blank.

4. Select the bottom view controller's tab bar item and set Title to Slide It and Image to Slider.

5. Select the tab bar item from the middle view controller and set Title to Action and Image to Alert.

Using the Asset Catalog and the storyboard, you've successfully named your tabs, set their icons, and created a relationship with the tab bar controller. The tabbed application is really starting to take shape. The visual relationship between the tab bar controller and the view controllers is in place, and if you want to, you can build and run the application in the simulator—it will work fine. However, there is one other relationship you've yet to establish.

The three views on the design area are currently controlled by the default view controller class. But you want to use the purpose-made view controllers that you created earlier in this practice.

1. Select the top bar above the Track It view controller, as in Fig. 11, and then open the Identity Inspector (+ +3).

2. Click the drop-down list for the Class attribute and select TrackViewController, as shown in Fig. 17.

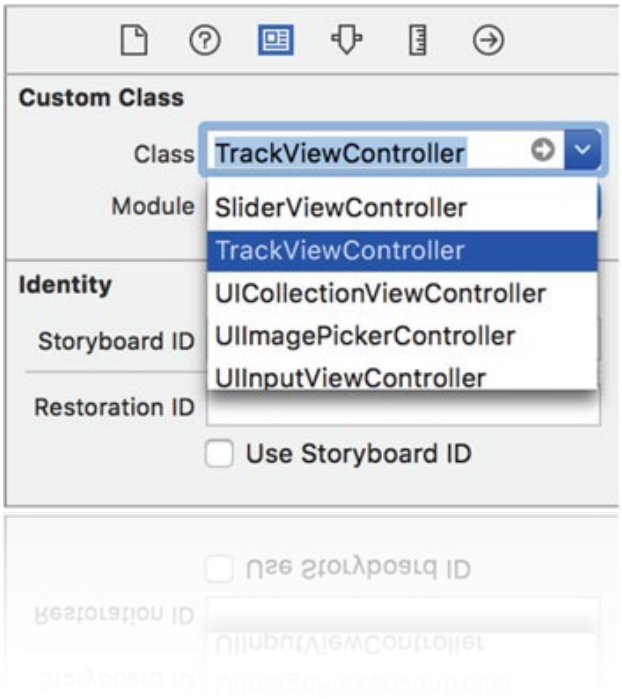

Fig. 17. *Selecting the custom view controller class*

3. Select the top bar above the bottom view controller, and this time set the class to SliderViewController.

4. Repeat Step 3 with the middle view controller and set its class to ActionViewController.

5. Build and run the application using the simulator; you should find that at this stage you have three bland but working tabs.

That's it: the preparation work is complete! So far you've renamed the default view controller classes and created an extra one, created entries in the Asset Catalog and populated it with some icons, removed the default view controllers from the storyboard and replaced them with three new ones all before setting the classes, icons, and titles of each tab. You're now ready to learn more about building great interfaces.

Tracking Location with the Track It Tab

For the first tab, you create a view that allows you to display detailed information about the current location, including speed, course, longitude, latitude, and positional accuracy. To do this, you will use the CoreLocation framework.

CoreLocation is used in many applications in the App Store, whether in an obvious way such as in a map-based application or in a more subtle way such as providing localized information wherever

you go. The skills you learn here will give you a good grounding in applying CoreLocation in your own applications.

CoreLocation by itself isn't that useful without something to control and display its information. To do this, you add a switch control to turn positional tracking on and off and a text view to display the output information. By the end of this section, you should have created something resembling the screen shown in Fig. 18.

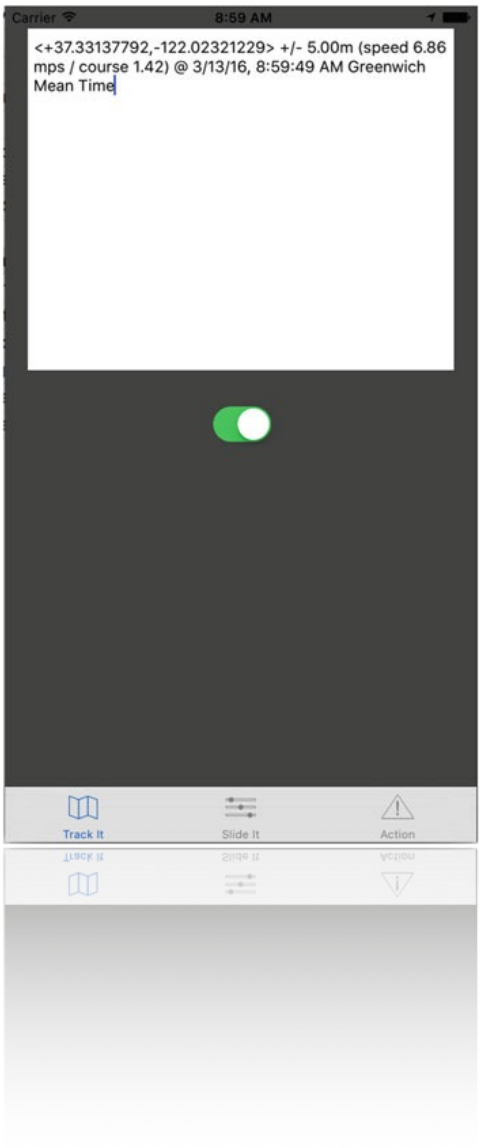

Fig. 18. *The Track It tab in action*

UITextViews and UISwitches

To build a simple but effective view, let's use two of the most common and useful controls provided by Apple: UITextView and UISwitch. The UISwitch control (or Switch, as it appears in the Object Library) is found throughout the Settings app on your iPhone or iPad. It has on and off states, and you use it to turn tracking on and off again:

1. Search the Object Library for Switch and drag the object to the view. Position it in the middle, as shown in Fig. 19. Blue guidelines appear as you approach the middle of the view.

Fig. 19. *Snapping the switch in to place using the guidelines*

2. With the switch selected, open the Attribute Inspector and change the State attribute to Off.

3. Add a UITextView (or Text View, as it appears in the Object Library). A text view can contain a large amount of text; the user can type in it, like a text field, or scroll through it. Search the Object Library for Text View and drag it so it appears just above the switch (refer back to Fig. 18 for size and positioning reference).

4. With the text view selected, open the Attributes Inspector and remove the placeholder text from the Text attribute. Next, uncheck the Editable behavior.

5. To change the view's background color, select a patch of whitespace on the view and then, in the Attributes Inspector, click the Background drop-down list and select Dark Gray Color from the list of specified colors to the right of the color indicator.

6. As you have in previous practice, with the view still selected, bring up the Fix Auto Layout Issues menu and click Add Missing Constraints under the All Views in View Controller

heading.

You've added the two controls needed for this tab. Next you need to create the actions and outlets within the TrackViewController class file that will allow your code to manipulate them:

1. Switch to the Assistant Editor and ensure that you have the TrackViewController.swift file selected. If you have a different file open, go back to double-check whether you correctly set the view controller's class.

2. Select the text view and control-drag a connection from the text view to the class file, just below the line that says class TrackViewController: UIViewController.

3. Create an outlet named locationText. Type this in and click the Connect button.

4. Repeat Step 3 for the switch, this time naming the outlet toggleSwitch.

5. Drag another connection from the switch, this time being sure to create an action, and name it changeToggle.

The code of your header file should look like this:

```
import UIKit
class TrackViewController: UIViewController {
   @IBOutlet weak var locationText: UITextView!
   @IBOutlet weak var toggleSwitch: UISwitch!
   @IBAction func changeToggle(_ sender: AnyObject) {
   }
   override func viewDidLoad() {
      super.viewDidLoad()
      // Do any additional setup after loading the view, typically from a nib.
}

   override func didReceiveMemoryWarning() {
      super.didReceiveMemoryWarning()
      // Dispose of any resources that can be recreated.
} }
```

You've created all your outlets and actions. But before you can write any code, you need to add the CoreLocation framework to the project so you can interact with the GPS features of your device.

▪ **Tip** If you accidentally create an outlet instead of an action you may have trouble running your application after removing the erroneous line. This is because your control is still looking for that outlet. Select the control and open the Connections Inspector. You can remove the reference to the nonexistent outlet there.

Adding Frameworks to a Project

A framework is a collection of classes and functions that provides additional functionality to your project. In iOS, all the GPS and location-based features are accessed through the CoreLocation framework. Some of the most commonly used frameworks are CoreData, MapKit, and CoreImage.

As of iOS 7 and Xcode 5, Apple gave Objective-C developers an alternative to manually adding frameworks, called *modules*. Modules are a very simple concept: instead of going through Xcode to select a framework, physically add it to a project, and reference it in code with an #import statement, you simply reference it with the @import statement, and Xcode automatically identifies the framework and adds it into your project.

In Swift, Apple has kept this functionality and made it the default approach. Thus you never again need to add a core Apple framework manually. As we've already mentioned, you need to add CoreLocation, and what have previously been a protracted process of locating and importing the framework is now as simple as writing a single line of code:

1. From the Project Navigator, select the TrackViewController.swift file and close the Assistant Editor in favor of the Standard Editor.

2. Drop down a line from import UIKit and type import CoreLocation. This single line makes the classes, functions, and protocols of the CoreLocation framework available to your application.

3. You need to specify that the view controller can act as a delegate for the CLLocationManager class. This means when the location manager is running, it knows this file contains the functions that handle certain events, such as the positionofthedevicechanging.Add, CLLocationManagerDelegateafterclass TrackViewController: UIViewController. As shown in Fig. 20, code completion appears to help you complete the protocol name.

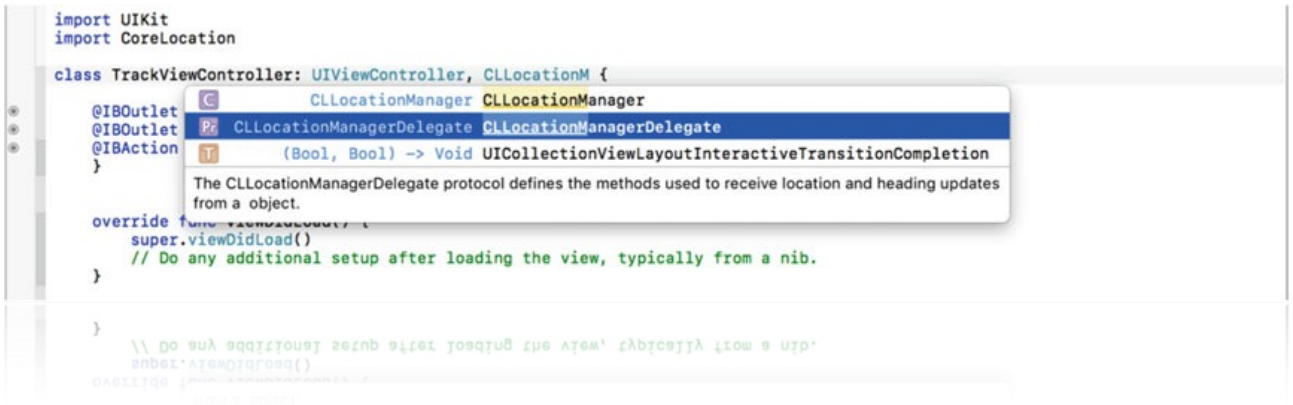

Fig. 20. *Adding the CLLocationManagerDelegate protocol to TrackViewController*

The first few lines of your TrackViewController.swift file should resemble the following code:

```
import UIKit
import CoreLocation
class TrackViewController: UIViewController, CLLocationManagerDelegate {
   @IBOutlet weak var locationText: UITextView!
   @IBOutlet weak var toggleSwitch: UISwitch!
   @IBAction func changeToggle(sender: AnyObject) {
}
```

4. You're ready to start setting up your interface into the device's GPS and location tracking technology, which is done via a class called CLLocationManager. Before the first @IBOutlet, add the following highlighted code to create an instance of the location manager class:

```
var locationManager: CLLocationManager!
@IBOutlet weak var locationText: UITextView!
```

▪ **Note** You have created the instance of the location manager in what is called the *global scope*; this means any of your functions in the TrackViewController class can see and use the location manager. This is important because the location manager is the primary interface into the location functions, and you want to be efficient and consistent by having only a single instance of the class declared in your application.

Next, you need to implement the action of the toggle switch being turned on or off. The code you add here does the majority of the work in this tab. A little below the line you just added should be the changeToggle action. Let's go through the code step by step before you see the final code block:

1. You need to determine whether the switch was turned on or off when the action is called. You do this with an if ... else ... statement. Add the highlighted code into the action:

```
@IBAction func changeToggle(_ sender: AnyObject) {
   if toggleSwitch.isOn {
} else {

}

}
```

The .on property of the UISwitch class returns a true or a false value, depending on the switch's position. If true or on, the code in the first set of braces is executed; otherwise, if false or off, the code in the second set of parentheses is executed.

▪ **Tip** In Swift, parentheses containing the conditions of an if statement are optional.

2. All the code you write in this view controller will be ignored if the device's location services are disabled. To account for this, the next block of code will prevent the switch from being turned on if location services are disabled. Add the highlighted code:

```
@IBAction func changeToggle(_ sender: AnyObject) {
   if toggleSwitch.isOn {
      if (CLLocationManager.locationServicesEnabled() == false) {
         self.toggleSwitch.isOn = false
```

```
}

} else {

}

}
```

3. The next step is to check whether the locationManager object has been initialized and, if not, to initialize it. There are numerous ways of initializing
a CLLocationManager object, but in this case you do four things: initialize the object, tell it that this view controller is acting as its delegate, tell it to be accurate within 10 meters, and tell it to update when it moves more than 10 meters from the last recorded position. So, drop down a line after the last statement and add the following highlighted code:

```
@IBAction func changeToggle(_ sender: AnyObject) {
   if toggleSwitch.isOn {
      if (CLLocationManager.locationServicesEnabled() == false) {
         self.toggleSwitch.isOn = false
}

      if locationManager == nil {
         locationManager = CLLocationManager()
         locationManager.delegate = self
         locationManager.distanceFilter = 10.0
         locationManager.desiredAccuracy = kCLLocationAccuracyNearestTenMeters
         locationManager.requestWhenInUseAuthorization()
}

} else {

} }
```

```
@IBAction func changeToggle(_ sender: AnyObject) {
   if(toggleSwitch.isOn)
   {
      if (CLLocationManager.locationServicesEnabled() == false) {
         self.toggleSwitch.isOn = false
}

      if locationManager == nil {
         locationManager = CLLocationManager()
         locationManager.delegate = self
         locationManager.distanceFilter = 10.0
         locationManager.desiredAccuracy = kCLLocationAccuracyNearestTenMeters
         locationManager.requestWhenInUseAuthorization()
}

locationManager.startUpdatingLocation()
```

```
} else {

} }

@IBAction func changeToggle(_ sender: AnyObject) {
   if(toggleSwitch.isOn)
   {
      if (CLLocationManager.locationServicesEnabled() == false) {
         self.toggleSwitch.isOn = false
}

      if locationManager == nil {
         locationManager = CLLocationManager()
         locationManager.delegate = self
         locationManager.distanceFilter = 10.0
         locationManager.desiredAccuracy = kCLLocationAccuracyNearestTenMeters
         locationManager.requestWhenInUseAuthorization()
}

      locationManager.startUpdatingLocation()
   }
else {

      if locationManager != nil {
         locationManager.stopUpdatingLocation()
}

} }
```

Those few lines can be used as boilerplate code any time you want to initialize a CLLocationManager.
We mentioned the didUpdateLocations delegate method that the locationManager object looks for every time an update is triggered. It's a very simple implementation that takes the last reported location information and outputs its description value to the text view. To do this, add the following highlighted code after the didReceiveMemoryWarning function:

```
override func didReceiveMemoryWarning() {
   super.didReceiveMemoryWarning()
   // Dispose of any resources that can be recreated.
}

func locationManager(_ manager: CLLocationManager, didUpdateLocations locations:
[CLLocation]) {
   let location:CLLocation = locations[locations.endIndex-1] as CLLocation
   self.locationText.text = location.description
}
```

You need to implement one final delegate method: the didFailWithError function that is called if there is a fault while trying to obtain a location and that writes the error description to the text view. It's not essential for this example, but we are trying to give you some useful boilerplate code; plus, you should always account for and handle failures such as this. Add the highlighted function below your last delegate function:

```
func locationManager(_ manager: CLLocationManager, didUpdateLocations locations:
[CLLocation]) {
    let location:CLLocation = locations[locations.endIndex-1] as CLLocation
    self.locationText.text = location.description;
}
func locationManager(_ manager: CLLocationManager, didFailWithError error: Error) {
    locationText.text = "failed with error \(error.localizedDescription) "
}
```

That's it—you've finished the code for the Track It tab. But before you run it, you need to do something new: add several entries to the application's info.plist file. For a number of frameworks and classes, Apple likes you to add a privacy declaration that explains to the users what you're doing with their location information. Since iOS 8, these are mandatory, and the code won't function without them:

1. In the Project Navigator, select Info.plist.

2. Move your mouse cursor over the first line, titled Information Property List. A small plus symbol appears, as shown in Fig. 21. Click it.

Fig. 21. *Adding an item to the application's info.plist file*

3. A new row is inserted. In the list on the left, scroll until you see the item Privacy - Location Usage Description. Select it, double-click the empty Value field, and type Testing CoreLocation or whatever message you want to present to the users. Your finished entry should resemble that shown in Fig. 22.

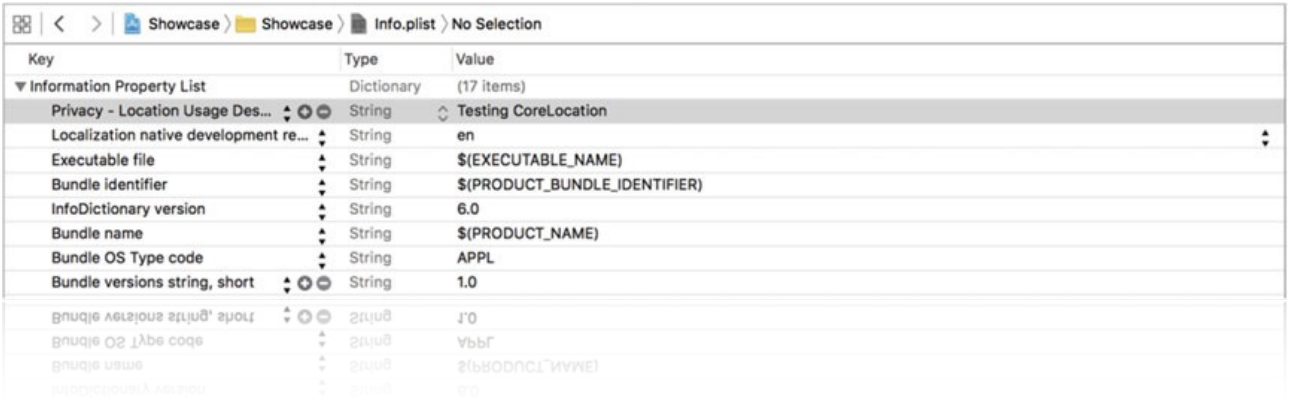

Fig. 22. *The privacy statement in the info.plist file*

4. Repeat Step 2 to create another entry under Information Property List. This time, you need to find the Privacy - Location When In Use Usage Description. Under Value, again enter Testing CoreLocation.

This may seem like an unnecessary chore, but without it, not only won't your app work, but it will also be rejected by Apple if you submit it to the App Store.

With the privacy message set, the last thing to do is test it in the simulator. When you flip the switch, you should see the privacy message, as shown in Fig. 23.

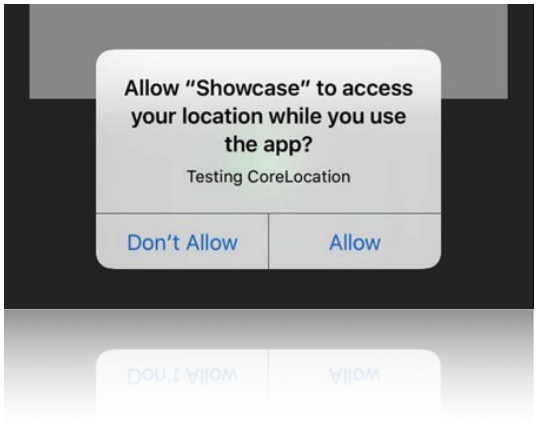

Fig. 23. *The custom privacy message being displayed to the users*

Simulating a Location

When you run your application in the Simulator and accept the privacy message, you may find that nothing happens. The reason for this is simple: by default, the Simulator doesn't have a location, and therefore it's unable to give you any details about a location, let alone update the location as it's moving.

▪ **Note** If nothing happens, you may not have Location Services enabled. Return to the home screen on your virtual device by going to Hardware ➤ Home, and then open the Settings app. In Privacy, select Location, and then ensure that Location Services is enabled.

Fortunately, Apple has provided some pretty nifty tools for specifying a location. It can also simulate a drive or bike ride, which is the preset you use in this case. In the simulator menu bar, choose Debug ➤ Location ➤ City Bicycle Ride. All of a sudden your text view begins filling as a virtual bike peddles through California, near Apple headquarters in Cupertino.

That does it for this tab! You've created a really neat app that you could deploy to your phone while you take a run to view your location and meters run per second reflected in real time, which is pretty amazing.

Mixing Colors with the Slide It Tab

The second tab uses UISlider controls to create a RGB (red green blue) color mixer that alters the background color in real time and outputs the values to a series of text fields. This is another tab with real- world, practical applications. RGB is a color system that defines colors by assigning three values between
0 and 255 to each primary color. Any web developer, graphic designer, or even iOS app programmer will at some point need a tool that gives them the RGB value for a certain color. With this tab, you can play around with different combinations before implementing the one you like.

The interface for this tab is by far the most complex of the three, so let's begin. You create one block of elements for the red color and then repeat the steps two times for the green and blue colors:

1. Add a Label object from the Object Library to the Slide It view. Position it near the top of the view and then double-click it and change the text to Red, as shown in Fig. 24.

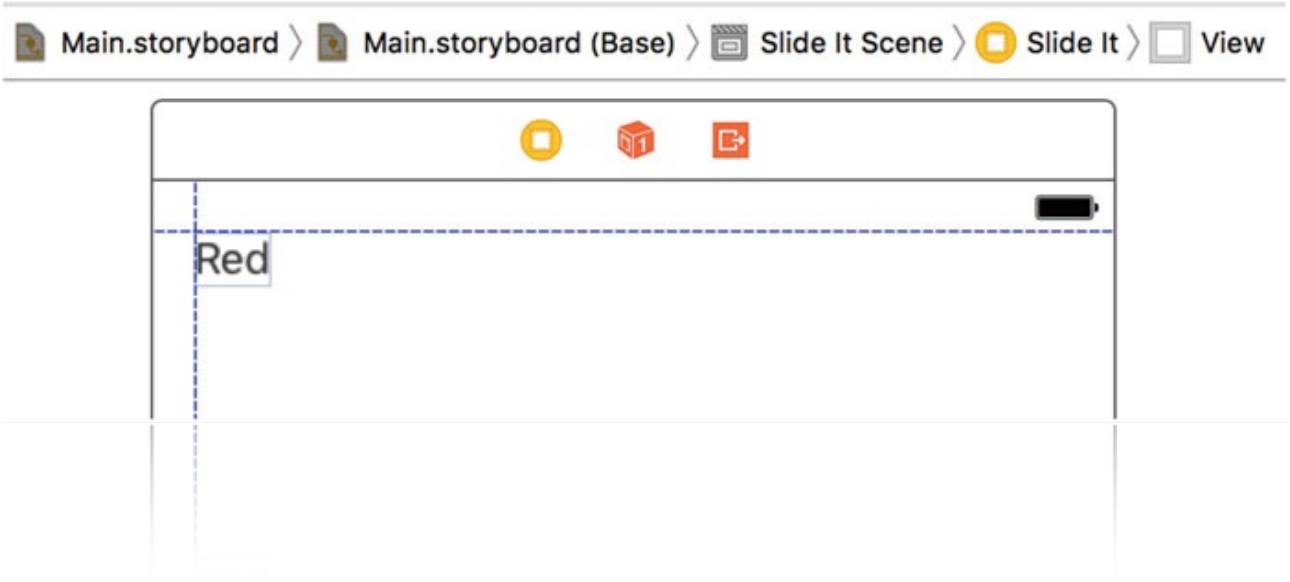

Fig. 24. *The color label in position*

2. Search for Slider in the Object Library and drag it onto the view. Position it below the label and resize it so it fills about two-thirds of the view's width, as shown in Fig. 25.

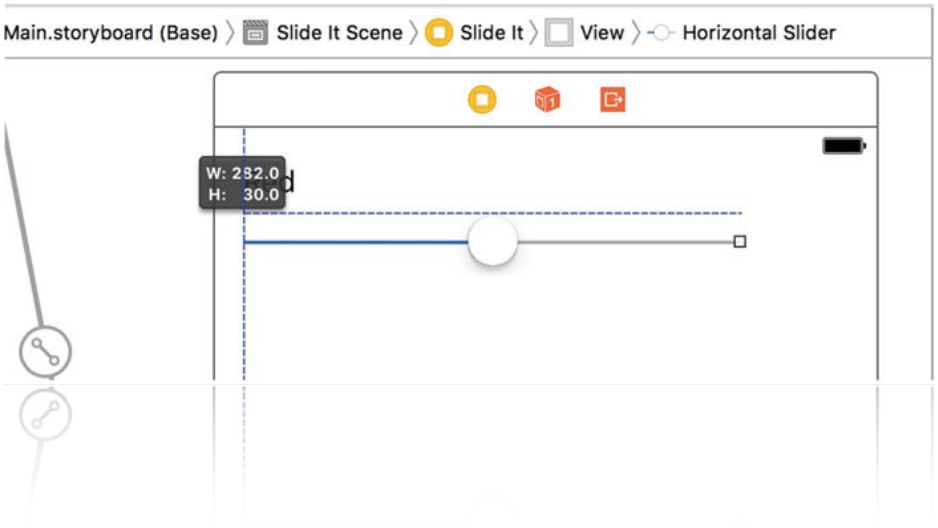

Fig. 25. *The slider added to the view and made wider*

3. You want to add a text field to display the RGB value. Drag in a text field from the Object Library and position it to the right of the slider, as shown in Fig. 26.

Fig. 26. *The text field added to the view and positioned to the right of the slider*

4. Repeat Steps 1 through 3, positioning each group of elements one under the other until your view resembles Fig. 27.

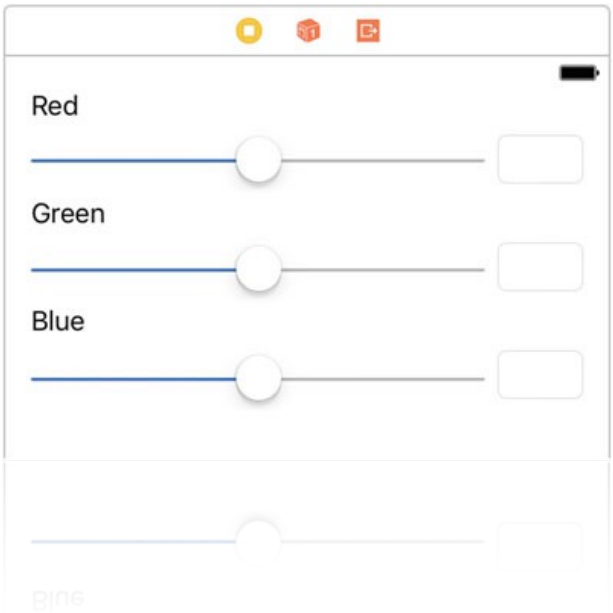

Fig. 27. *The finished interface*

5. To ensure that all the elements line up when you run the application, take a moment to add the constraints for the view. Select a white area of the view, click the Fix Auto Layout Issues button, and click Add Missing Constraints.

6. Select the red slider and examine its values in the Attributes Inspector. Its value range is set with 0 as a minimum value and 1 as a maximum. Your instinct might be to set the maximum to 255, the upper value of a color in the RGB format, but the class responsible for creating colors expects a value between 0 and 1, so this fits your needs perfectly. You do, however, want to change the starting point for the slider to be the maximum value, so change the value to 1 from 0.5. Repeat this for the green and blue sliders.

This completes the interface, leaving you ready to create your outlets and actions before moving on to the code, which is very simple for this tab. As you did in the previous tab, open the Assistant Editor and ensure that it shows the SliderViewController.swift file:

1. Create an outlet for each of the UISlider controls, naming them redSlider, greenSlider, and blueSlider, respectively.

2. Create outlets for each of the UITextFields, naming them redValue, greenValue, and blueValue, respectively.

3. Create actions for all the UISlider controls, naming them changeRed, changeGreen, and changeBlue, respectively.

4. For reasons that we go into shortly, make your view controller a text field delegate by adding , UITextFieldDelegate after class SliderViewController: UIViewController.

Before you move on, check that the start of your code looks like this:

```
import UIKit
class SliderViewController: UIViewController, UITextFieldDelegate {
    @IBOutlet weak var redSlider: UISlider!
    @IBOutlet weak var greenSlider: UISlider!
    @IBOutlet weak var blueSlider: UISlider!
    @IBOutlet weak var redValue: UITextField!
    @IBOutlet weak var greenValue: UITextField!
    @IBOutlet weak var blueValue: UITextField!
    @IBAction func changeRed(_ sender: AnyObject) {
    }
    @IBAction func changeGreen(_ sender: AnyObject) {
    }
    @IBAction func changeBlue(_ sender: AnyObject) {
    }
```

That's it for Interface Builder for this tab. This has been one of the most complex interfaces you've encountered so far. Switch back to the Standard Editor and open SliderViewController.swift from the Project Navigator:

1. As with the previous tab, you need to store the value specified by the sliders by declaring and initializing some global variables that are of CGFloat
type. Add the following code after the line class SliderViewController: UIViewController, UITextFieldDelegate {:

```
var redColor:CGFloat = 1.0
var greenColor:CGFloat = 1.0
var blueColor:CGFloat = 1.0
```

2. Navigate to the viewDidLoad function. Under the line super.viewDidLoad(), you need to set the delegate property of the text fields in order to use the UITextViewDelegate protocol. Add these lines:

```
redValue.delegate = self
greenValue.delegate = self

blueValue.delegate = self
```

3. You're going to call a function that you haven't written yet, so don't panic when Xcode doesn't help you through code completion and then adds
a red exclamation mark next to this line. You call the function by adding updateColor() to the viewDidLoad function after the last code you wrote. Your completed viewDidLoad function should now look like this:

```
override func viewDidLoad() {
    super.viewDidLoad()
    redValue.delegate = self
```

```
    greenValue.delegate = self
    blueValue.delegate = self
    updateColor()
}
```

4. You need to write the updateColor function, which takes the red, green, and blue values and uses them to set the view's background color. Under the viewDidLoad function, add the following code:

```
func updateColor() {
    self.view.backgroundColor =
UIColor(red: redColor, green: greenColor, blue: blueColor, alpha: 1.0)
}
```

In this code, you create a UIColor object from the red, green, and blue values. The alpha property controls the opacity of the background, with 1.0 being totally opaque and 0.0 being transparent.

Now you need to add the code to the three actions that are linked with their corresponding sliders: changeRed, changeGreen, and changeBlue. All of these actions use practically the same code—only the variable and outlet names change, depending on the color. Let's set the changeRed code step by step, after which you complete the remaining two methods yourself:

1. Take the value from the slider and assign it to the redColor float. In the action for theredslider,simplywriteredColor = CGFloat(redSlider.value).

2. You want to update the text field with the correct RGB value. To do that, you need to convert the value from between 0.0 and 1.0 to between 0 and 255, so, you multiply the value of redColor by 255. Finally, you ensure that there are no decimal places by using the String(Format: function and the %.0f placeholder, which in plain English means "put the float value here but limit it to 0 decimal places." The number before f controls the number of decimal places shown
in the string. Also, in order to make the format function recognize the float, you need to convert it from a CGFloat to a Float. The code to achieve this is redValue.text = String(format: "%.0f",Float(redColor*255.0)).

3. A change has been made, so you need to call the updateColor function to make sure the change is reflected in the color set in the view's background. The code for this is the same as in the viewDidLoad function, so type updateColor().

The code for the finished action should look like this:

```
@IBAction func changeRed(sender: AnyObject) {
    redColor = CGFloat(redSlider.value)
    redValue.text = String(format: "%.0f",Float(redColor*255.0))
    updateColor()
}
```

Your challenge is to implement the remaining two actions by yourself. When you're done, check that your code matches mine:

```
@IBAction func changeGreen(sender: AnyObject) {
    greenColor = CGFloat(greenSlider.value)
    greenValue.text = String(format: "%.0f",Float(greenColor*255.0))
    updateColor()
}
@IBAction func changeBlue(sender: AnyObject) {
    blueColor = CGFloat(blueSlider.value)
    blueValue.text = String(format: "%.0f",Float(blueColor*255.0))
    updateColor()
}
```

You need to write one final function to complete this tab: the textFieldShouldReturn method, which

the text fields will look for now that they know this view controller is acting as a delegate for those text fields.

The UITextViewDelegate Implementation

Text fields are probably the most common control in an iOS app—they're everywhere. You tap inside them, the keyboard slides in, and you add your text. It's probably second nature to you that tapping the Return key dismisses the keyboard. Hold that thought; go ahead and run your application and select the Slide It tab.

Play around with the sliders and see how the background color changes as you modify the values. You've created something that can be usefully applied in the real world, which, as we mentioned previously, is done by giving the RGB values so they can be selected. Let's test this. Tap in one of the text fields: as expected, the keyboard slides in. Great—now try to go to the Track It tab. Hmm, not so great: the keyboard is blocking the path, so you're effectively stuck and must quit and relaunch the app to have any hope of accessing the other tabs.

You want to make it so that when you press Return, the keyboard dismisses itself. This is why you made your view controller take on the UITextViewDelegate role. By doing this, when you press Return, the text field tries to call the textFieldShouldReturn function; but because you haven't added this function yet, it doesn't do anything. Add the following code beneath your viewDidLoad function:

```
func textFieldShouldReturn(textField: UITextField) -> Bool {
    textField.resignFirstResponder()
    return true
}
```

When you tap the text field, it assumes responsibility for everything that happens thereafter—in other words, it becomes the *first responder*. When this function is called, you're telling it to give up this status with the resignFirstResponder function before returning a Boolean value, which in this case can be true or false (the result is the same). Rerun your application: you should find that you can dismiss the keyboard with the Return key and that you have a fully functional color slider view, as shown in Fig. 28.

▪ **Note** If you're running this in the simulator, as of Xcode 6 and iOS 8, the keyboard doesn't automatically show—the simulator assumes you want to use your physical keyboard. But on a device, you experience the problem of not being able to dismiss the

keyboard, which is why you must always test on a physical device before releasing to the App Store. To summon the keyboard in the simulator, go to Hardware ➤ Keyboard ➤ Toggle Software Keyboard (+K).

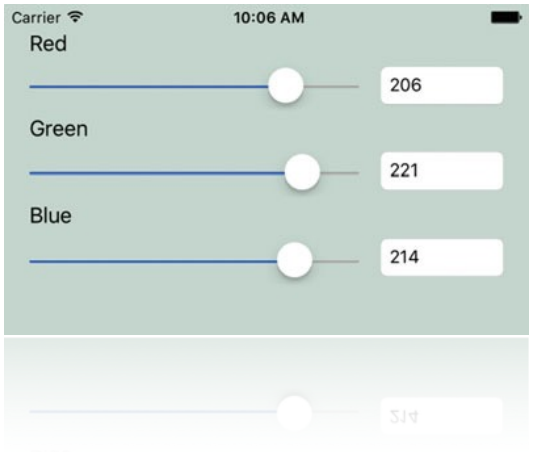

Fig. 28. *The finished Slide It tab, complete with dismissible keyboard*

Adding “Off the Menu” Controls

You’ve created two hugely different but incredibly cool tab views so far. For the third tab, you look at another common control you add through Interface Builder: the segmented control. In addition, you look at two important controls that you can’t add through Interface Builder: the alert view and the action sheet.

Alert Views and Action Sheets with UIAlertController

Before you begin building your interface, let’s clarify what we mean when we talk about alert views and action sheets. Fig. 29 shows how both are used in the iCloud settings area of the Settings application in iOS 8. You already encountered an alert view, when the Track It tab asked for access to your location.

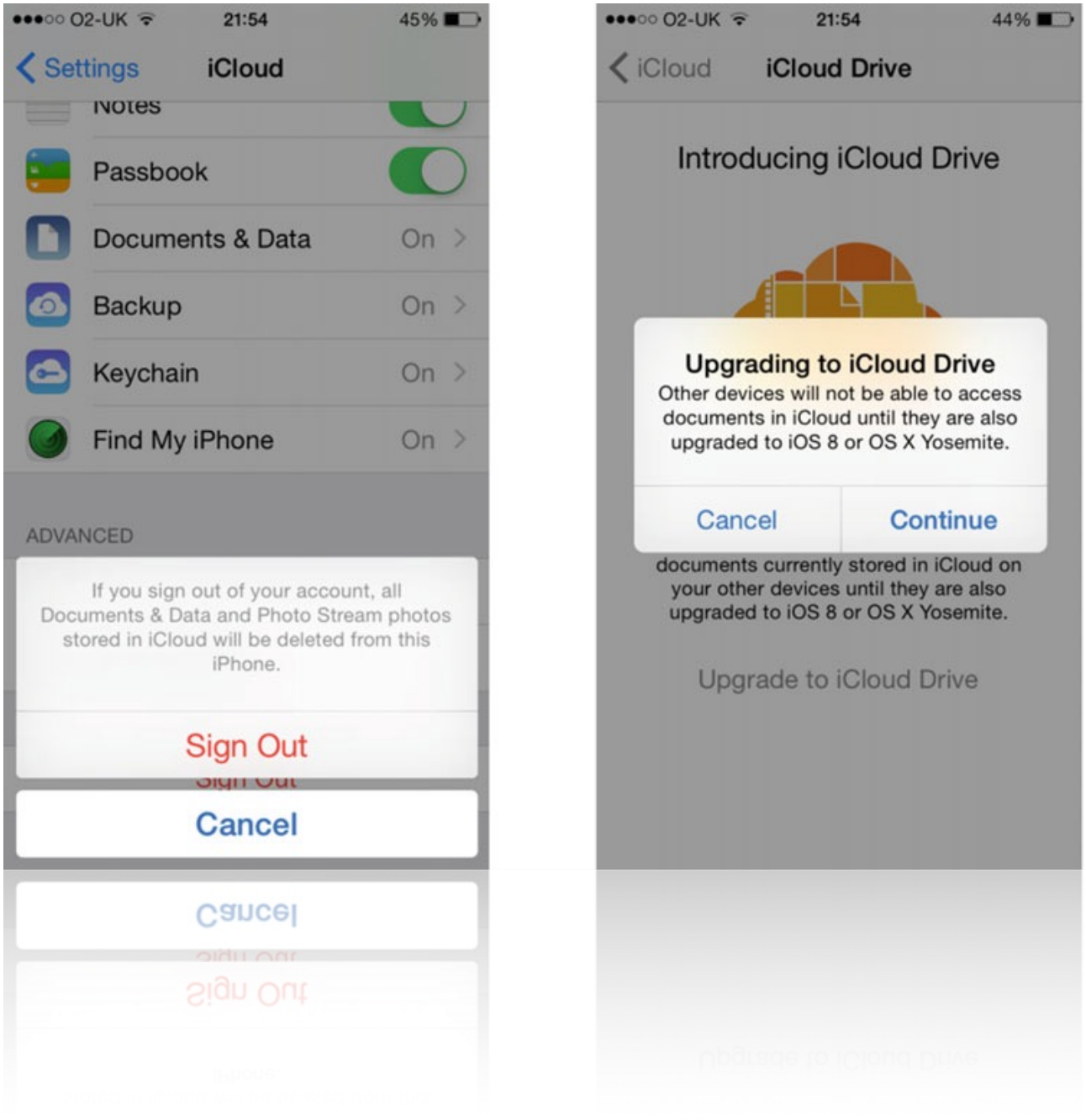

Fig. 29. *An example of an action sheet (left) and an alert view (right)*

Action sheets, as their name implies, can be used to present the users with several options for a specific action. For example, when you tap the flag icon while looking at an e-mail, you're asked whether you want to Flag, Mark as Read/Unread, or Move To Junk. If you give the users the option to add account details to your application, you might use an action sheet to ask whether the users want to add an account for your site or a third-party account, such as an OpenID account.

Alert views are coded in a way very similar to action sheets. Alert views are used to draw user attention to an event, such as a timer ending, or to confirm whether users want to activate a feature or delete some data. You'll use them often, and the good news is that they're easy to set up and use.

In iOS 8, Apple introduced a new class called UIAlertController, which combines the older UIAlertView and UIActionSheet classes into a single class. This is a fairly sensible move on Apple's part; the legacy classes were almost identical in syntax.

Building the Action Tab Interface

Now that you have a clearer understanding of what alert views and action sheets do, you're ready to build the third and final tab: the Action tab. Just the middle-right view controller remains to be built. Adjust your storyboard so it's visible in the design area:

1. Search for Segmented Control in the Object Library, as shown in Fig. 30. Drag it onto the view and position it in the center, at the top of the view.

Fig. 30. *Searching for Segmented Control in the Object Library*

2. Change the values of the segments by selecting the segmented control you added to the view. Open the Attributes Inspector. Change the segment Title attribute from First to Alert and press return.

3. Changing the second segment's title isn't as obvious. Looking at the Attributes Inspector for the segmented control, notice the drop-down list above the Title attribute that you just changed. Click it and select Segment 1- Second.

4. You can now change the Title attribute of the second segment from Second to Action, as shown in Fig. 31.

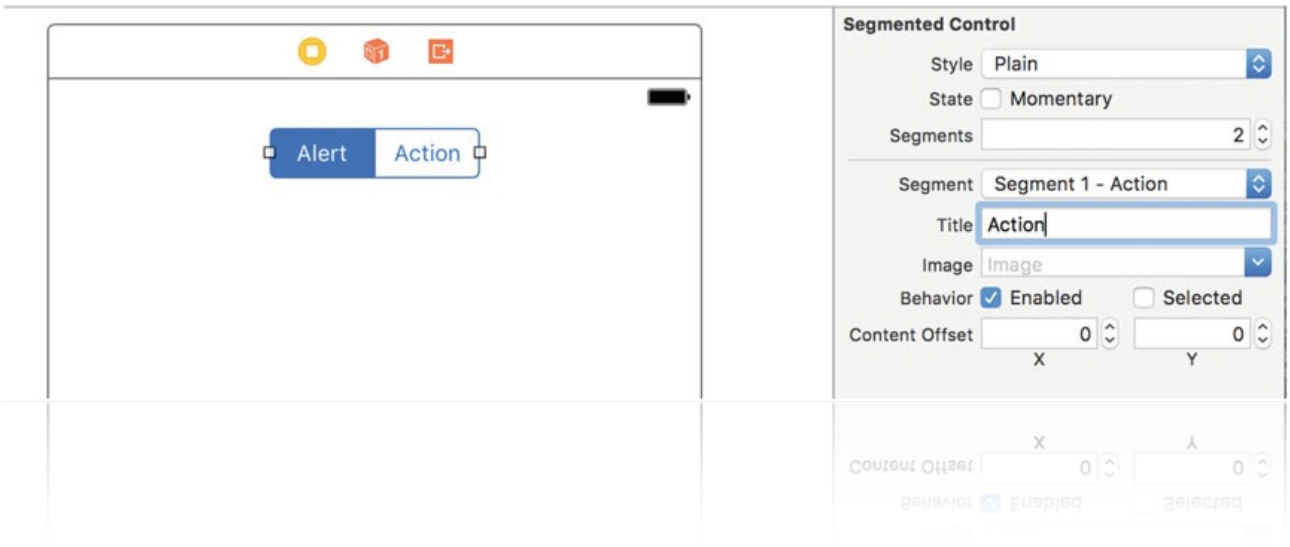

Fig. 31. *Changing the second segment's Title attribute*

5. Using the square handles on either side of the segmented control, resize it so that you can see all of the text in the second segment. Reposition it so it's centered again.

6. Add a button control to the view to trigger whichever option is selected. Search for Button in the Object Library, and drag it on to the view, positioning it dead center in the middle of the view as you did with the switch in the Track It tab.

7. Using the Attributes Inspector, change the button's Title attribute from Button toShow Me.Again,youneedtorepositionittobedeadcenterafterchangingthe text. Your view should resemble Fig. 32.

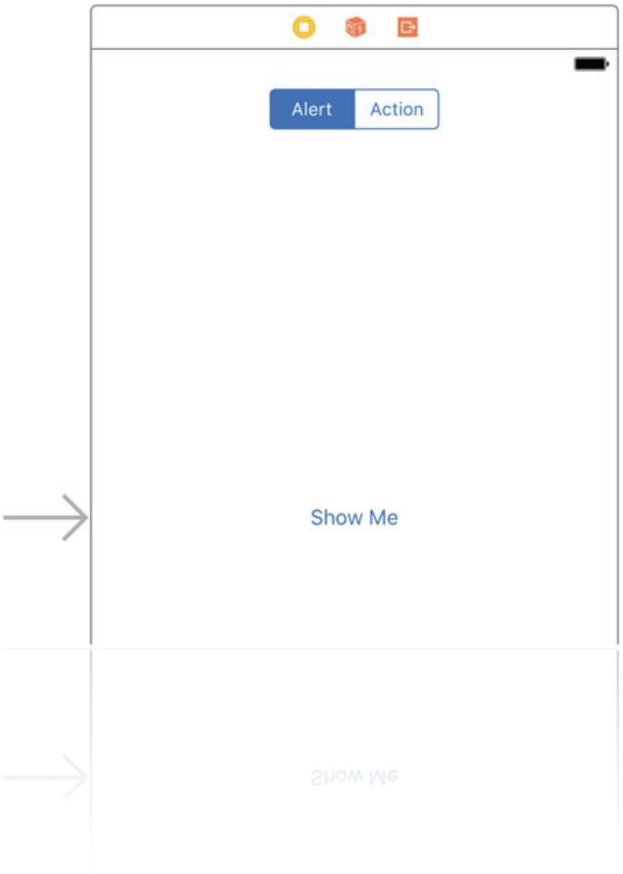

Fig. 32. *The completed interface for the Action tab*

8. Click the view, go to Fix Auto Layout Issues, and click Add Missing Constraints.

9. You're now ready to create the outlets and actions. As usual, switch to the Assistant Editor and ensure that you have ActionViewController.swift selected. Control-drag a connection from the segmented control into the header and create an outlet named actionControl, as shown in Fig. 33.

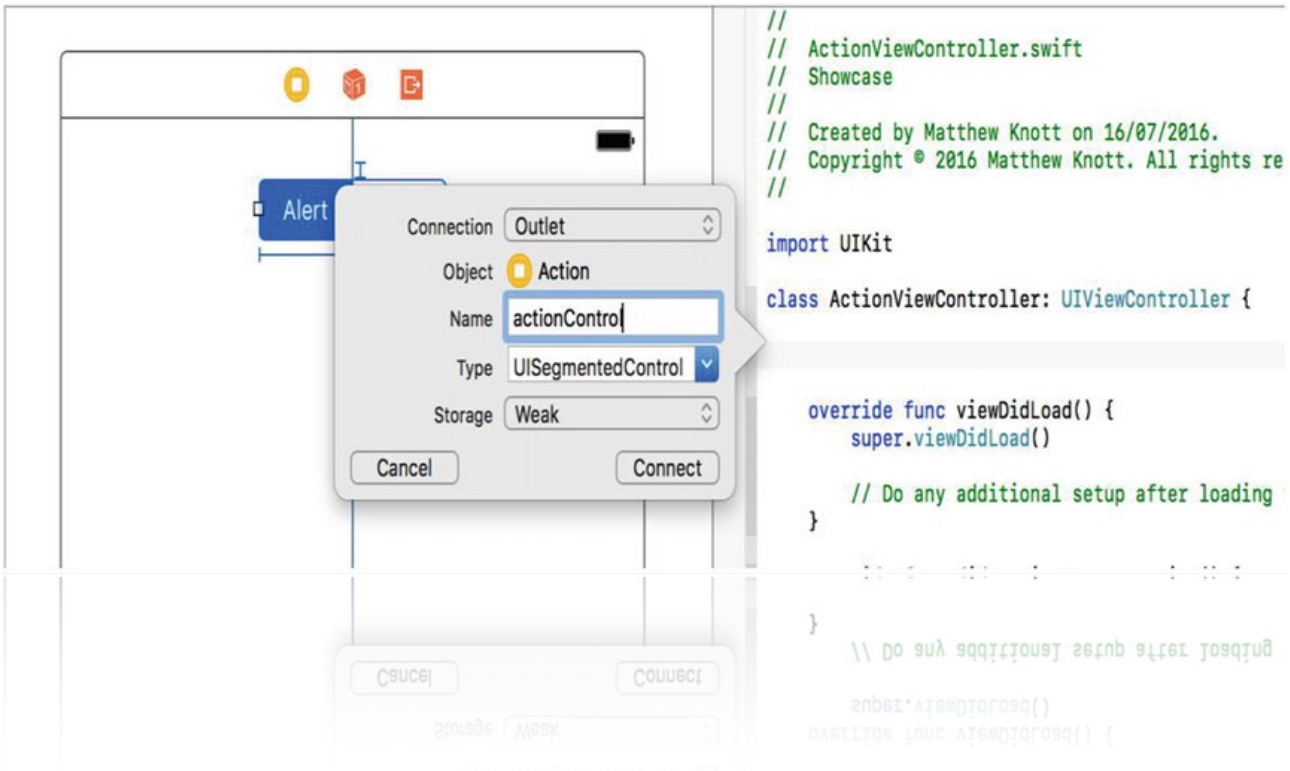

Fig. 33. *Creating the action Control outlet*

10. Create an outlet for the button called showmeButton, and then create an action

for it named performAction.
The first lines of your ActionViewController.swift file should now resemble the following code:

```
import UIKit
class ActionViewController: UIViewController {
    @IBOutlet weak var actionControl: UISegmentedControl!
    @IBOutlet weak var showmeButton: UIButton!
    @IBAction func performAction(_ sender: AnyObject) {
    }
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }
```

It's important to check the class documentation when experimenting with different controls and frameworks, because quite often you need to specify that your view controller is acting as a delegate for the classes you're adding. This can be the case with action sheets and alert views if you want to take advantage of any of their delegate methods for handling user responses. Missing a delegate reference can lead to your application failing or your code not being called in some situations. Because you won't be using the delegate methods in this example, there is no need to add them.

You're now ready to begin coding the action in this class file. Switch back to the Standard Editor and open ActionViewController.swift from the Project Navigator. All you need to look at in the file is the stub for the performAction action:

1. Scroll down until you find the performAction action. Inside its braces, you'll typeanif ... else ...statementtoseewhichsegmentiscurrentlyselected and determine the appropriate action to perform. You do this by checking the UISegmentedControl's selectedSegmentIndex property. The segments are held in an array, and the index is an incremental number assigned to each entry. The index starts at 0, so if the selected index is 0, that means the alert is selected; if it's 1, that means the action sheet is selected. Type the highlighted code into the action:

```
@IBAction func performAction(_ sender: AnyObject) {
   if actionControl.selectedSegmentIndex == 0 {
} else {

}

}
```

2. You need to initialize and show the alert view. The new UIAlertController takes far more code to initialize than its predecessor, but it's far more flexible. Type the highlighted code; once the action sheet code is written, you can see the completed result:

```
@IBAction func performAction(_ sender: AnyObject) {
   if actionControl.selectedSegmentIndex == 0 {
      let controller : UIAlertController = UIAlertController(title: "This is an alert",
         message: "You've created an alert view",
         preferredStyle: UIAlertControllerStyle.alert)
      let okAction : UIAlertAction = UIAlertAction(title: "Okay",
         style: UIAlertActionStyle.default,
         handler: {
            (alert: UIAlertAction!) in controller.dismiss(animated: true, completion:
nil) })

      controller.addAction(okAction)
      self.present(controller, animated: true, completion: nil)
} else {

} }
```

3. To code the else eventuality, type this very similar highlighted code inside the second set of parentheses:

```
@IBAction func performAction(_ sender: AnyObject) {
   if actionControl.selectedSegmentIndex == 0 {
      let controller : UIAlertController = UIAlertController(title: "This is an alert",
         message: "You've created an alert view",
         preferredStyle: UIAlertControllerStyle.alert)
      let okAction : UIAlertAction = UIAlertAction(title: "Okay",
         style: UIAlertActionStyle.default,
         handler: {
(alert: UIAlertAction!) in controller.dismiss(animated: true, completion: nil)

})

      controller.addAction(okAction)
```

```
        self.present(controller, animated: true, completion: nil)
    }
else {

        let controller : UIAlertController = UIAlertController(title: "This is an action
        sheet",
            message: "You've created an action sheet",
            preferredStyle: UIAlertControllerStyle.actionSheet)
        let okAction : UIAlertAction = UIAlertAction(title: "Okay",
style: UIAlertActionStyle.default,
    handler: {
        (alert: UIAlertAction!) in controller.dismiss(animated: true, completion:
nil) })

controller.addAction(okAction)
self.present(controller, animated: true, completion: nil)
} }
```

You've just written code that performs four distinct tasks. First, you define a UIAlertController called controller. Next, you define a UIAlertAction that adds a button to either the alert view or the action sheet to dismiss the controller. Third, you add the action to the controller, associating the two. Finally, you tell the main view to present the UIAlertController. The only difference between these two pieces of code is that UIAlertControllerStyle is alert for an alert view and actionSheet for an action sheet.

That's it—you've configured your view controller to show either an alert view or an action sheet depending on the selected index of a segmented control.

You've coded the third and final tab, so go ahead and run the application in the simulator. It should produce results similar to those shown in Fig. 34. Look at all the great things you've been able to achieve in this practice, with a relatively small amount of effort and code! Hopefully your confidence with the Xcode IDE, iOS, and the Swift language is beginning to build.

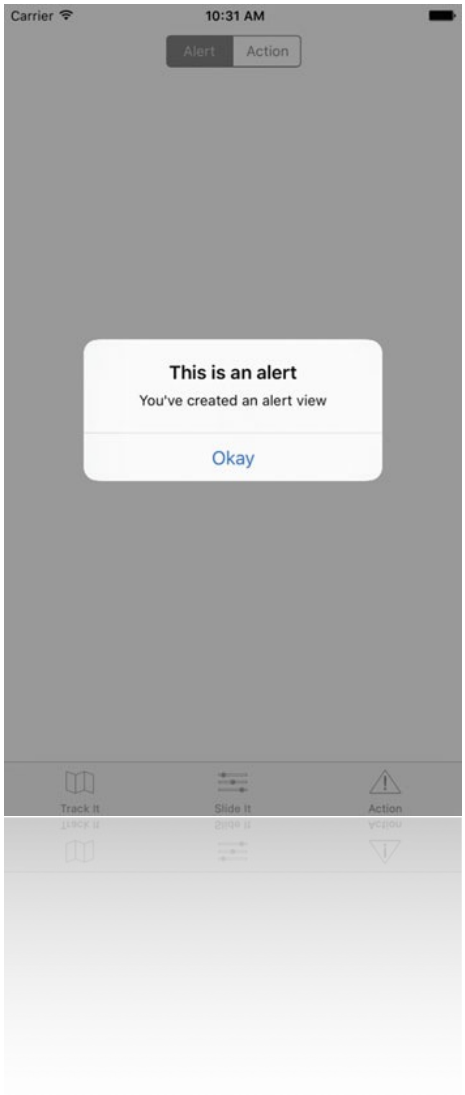

Fig. 34. *The alert view shown in the Alert tab*

Changing the Interface with Code

In this practice, you've taken a good look at how you can adjust the interface elements' attributes using the Attribute Inspector. But just as you can't use Interface Builder to add action sheets and alert views, there are some visual effects that you can only achieve through code. You've already done a lot of hard work in this practice, and you won't learn any more about Xcode here. So, look at this section as totally optional. However, you'll probably want to use the skills you can develop here to build your own applications for iOS devices, in which case these examples will prove invaluable.

Styling Buttons

With iOS 7, Apple introduced the most radical change in design since the launch of the first iPhone: moving away from skeuomorphism to a flat design style. The decision was controversial when announced, but many are now warming to the change and have adapted their applications to fit with the new styles.

One area that changed that many want to alter in their applications is the standard button. Fig. 35 shows the three buttons from the Contact screen in iOS 6 and iOS 7. In iOS 6, buttons looked like traditional buttons, whereas in iOS 7 onwards, they're shown in the same style as hyperlinks on a web page.

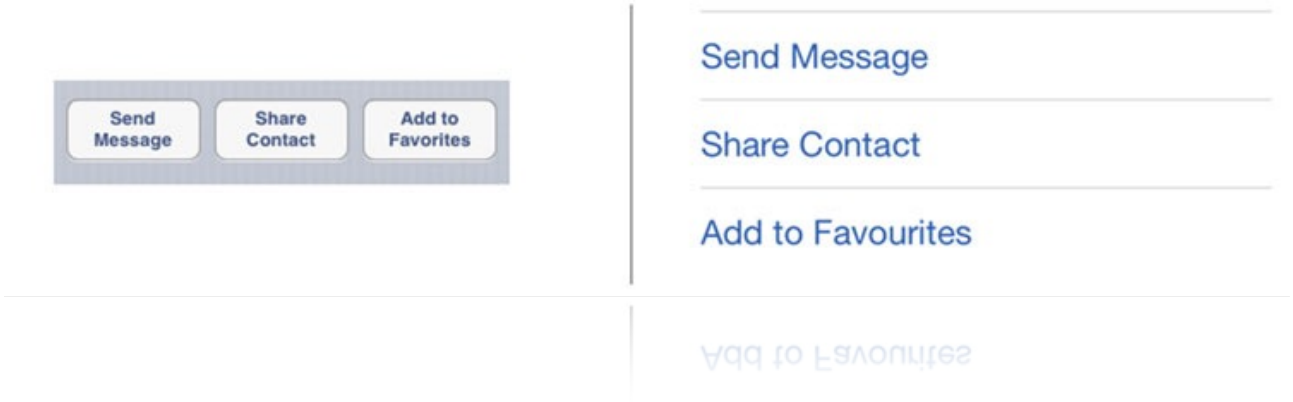

Fig. 35. *The three buttons from the Contact app's detailview for iOS6 (left) and updated for iOS7 (right)*

Although you can change the background color of the button, you can't add rounded corners in

Interface Builder, so you need to delve into code to make these alterations:

1. From the Project Navigator, open ActionViewController.swift and scroll down to the viewDidLoad function. Drop down a line after super.viewDidLoad(), and you're ready to add some custom code.

2. You're going to change the background color to a dark blue. You do this similarly to how you changed the background color in the Slide It tab, by creating a color using RGB values. But this time, you need to convert real RGB values, which range from 0 to 255, to fit in with what the method expects, which is a value between 0.0 and 1.0. To do this, you divide the value by 255.0. Add this line of code:

```
showmeButton.backgroundColor =
        UIColor(red: 9/255.0, green: 95/255.0, blue: 134/255.0, alpha: 1.0)
```

3. The button will be hard to read with blue text on a blue background, so the next task is to change the text color to white. You could do this in Interface Builder, but then you wouldn't be able to read the button's text when looking at the storyboard. Type the following code on the next line:

```
showmeButton.setTitleColor(UIColor.white, forState: UIControlState.normal)
```

4. You can easily apply a curved corner to the button by specifying a float value greater that 0.0 to the button's cornerRadius property. You do this using the following code:

```
showmeButton.layer.cornerRadius = 4.0
```

Your viewDidLoad function should now look like this:override func viewDidLoad() {

```
    super.viewDidLoad()
    showmeButton.backgroundColor =
        UIColor(red: 9/255.0, green: 95/255.0, blue: 134/255.0, alpha: 1.0)
    showmeButton.setTitleColor(UIColor.white, for: UIControlState.normal)
    showmeButton.layer.cornerRadius = 4.0
}
```

Run the application in the simulator: you see the difference immediately in your button on the Action

tab. The problem is that the button isn't set at a suitable size to make the most of your effects.

1. Open Main.Storyboard from the Project Navigator, make the button on the Action view much bigger, and then reposition it to the center.

2. Click the Resolve Auto Layout Issues button and choose Reset to Suggested Constraints.

3. Now run the application again: your button should look great and resemble that shown in Fig. 36.

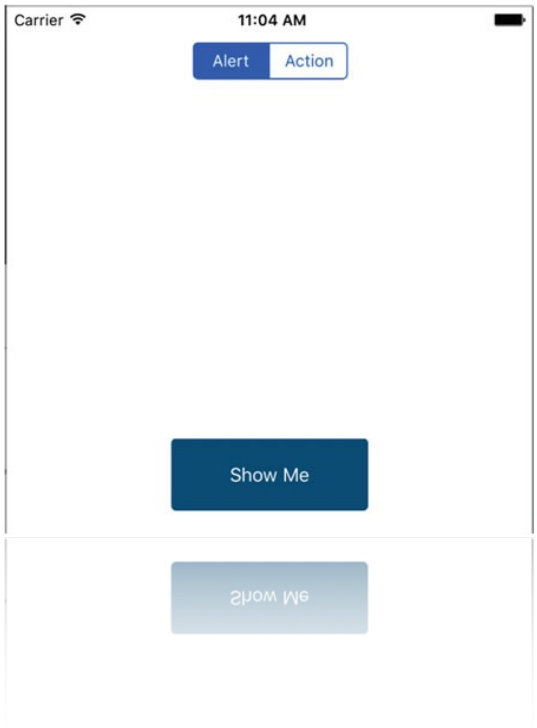

Fig. 36. *The customized button*

That's it for this practice. As a final challenge, try to apply curved corners to the text view in the Track It tab using the code you used to curve the button. If you get stuck, the answer can be found at the end of the summary for this practice.

Summary

This has been a long practice, but you've made it through and should be really proud of what you've achieved. The objective of the practice was to learn more about creating interfaces in Xcode, and you did that with a mix of Interface Builder and writing custom code. The application you created was called Showcase, mostly because it gives you a cool app that you can load on your phone so you can show your friends and colleagues the kind of amazing things you're now able to develop!

Specifically, in this practice, you did the following:

- Created an application from the Tabbed Application template
- Renamed the default view controllers and created your own from scratch
- Removed the default views from the storyboard and created three of your own
- Tied your new view controllers to their respective classes
- Created image sets in the images Asset Catalog and populated them
- Linked views to a tab bar controller in a storyboard
- Learned about frameworks and accessed the device's GPS function
- Learned about the UITextView, UISegmentedControl, UISwitch, and UISlider controls
- Programmatically created an alert view and an action sheet using UIAlertController
- Learned how to modify the visual appearance of controls using code

When you go through that list, you can see how many new skills you've learned in this practice. Before moving on, though, we promised the solution to rounding the corners of the text view in the Track It tab. If you did it right, you should have added the following line to the viewDidLoad function in TrackViewController.swift:

```
locationText.layer.cornerRadius = 5.0
```

***