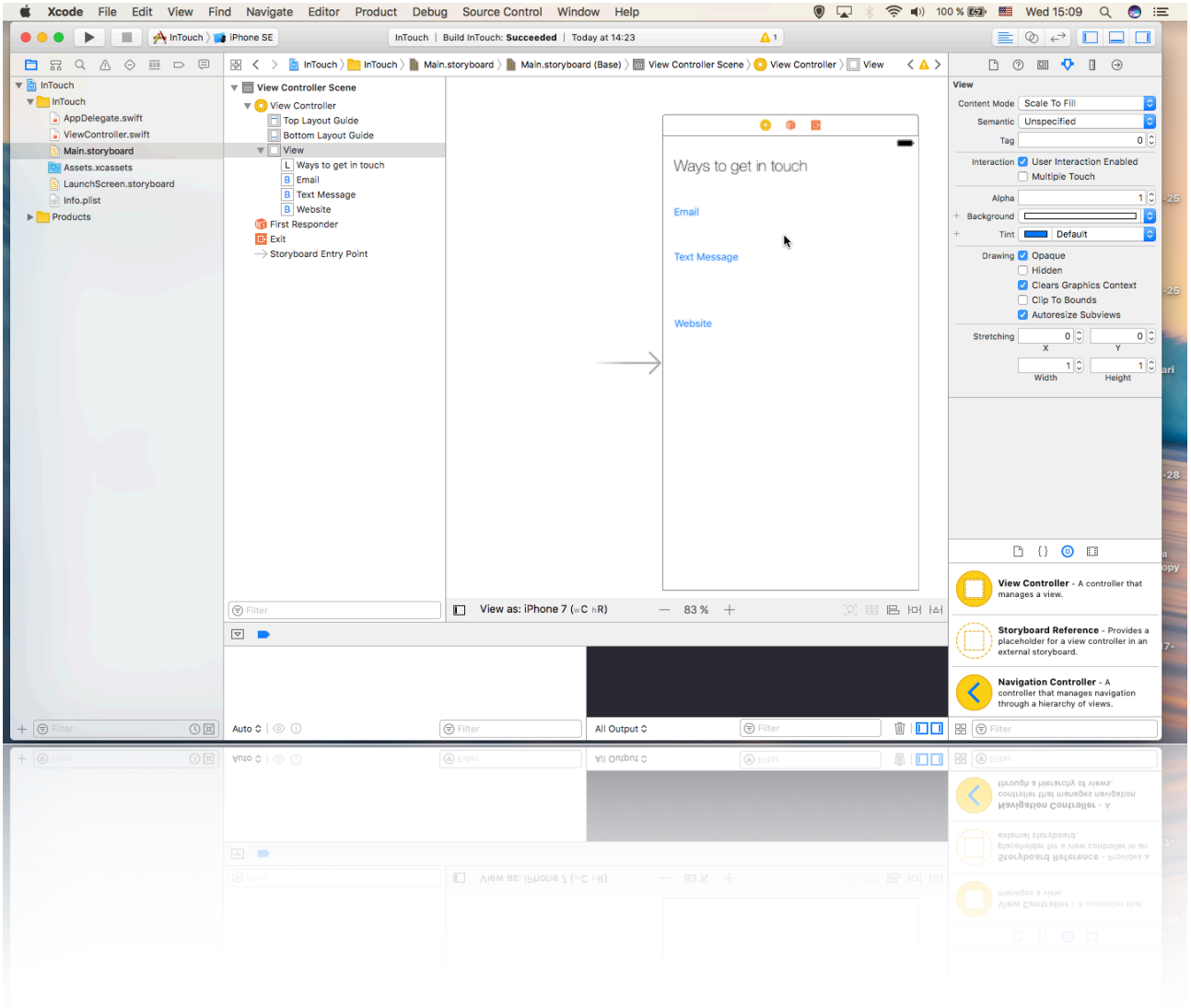
InTouch App

Practice in Xcode with Swift (tutorial #3)

This practice explains how to create the app, which demonstrates how to interact with some of the built-in applications: Mail, Messaging, and Safari.



Creating the Project

Okay, let's start building the project.

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

2. Name your project InTouch and ensure that Language is set to Swift and the Devices option is set to Universal. Configure the other settings as you did in the previous applications. Make sure the key settings match those shown in Figure 5-2. Click Next.

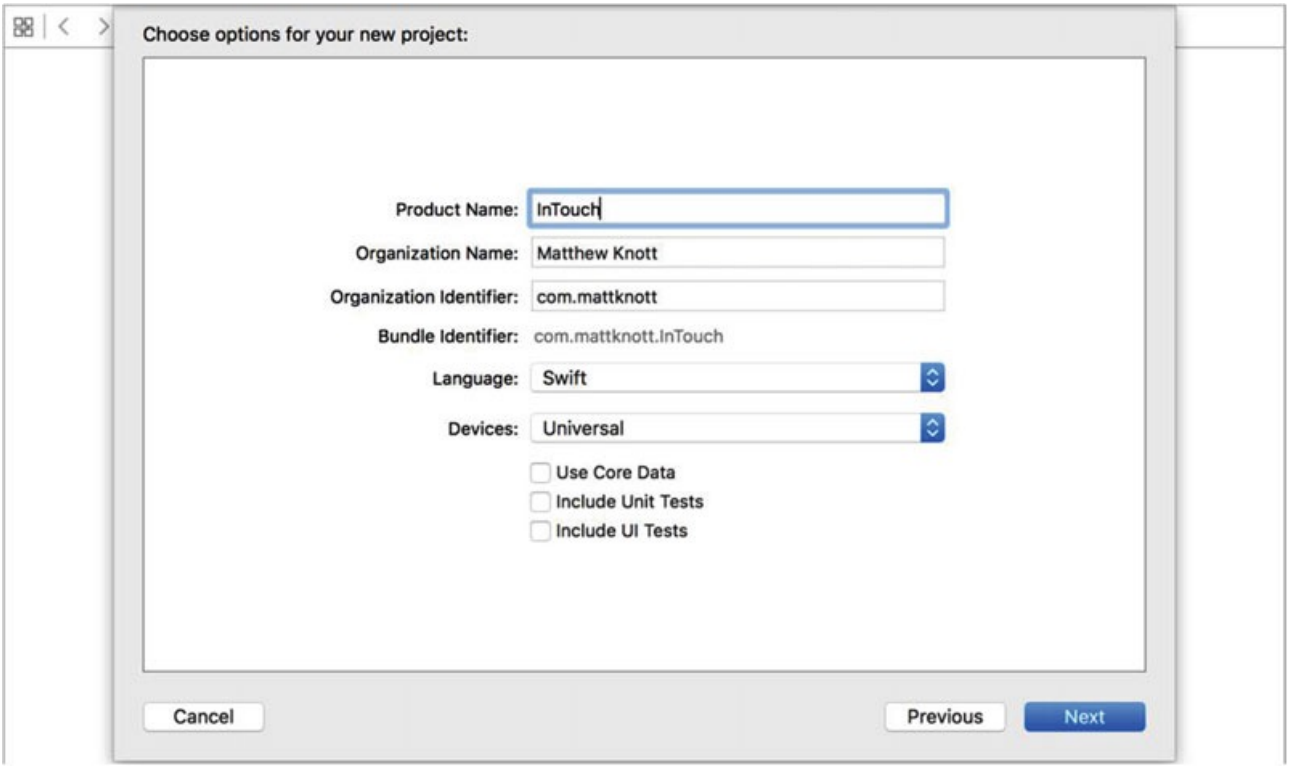


Figure 5-2. *The initial settings for the InTouch application*

3. You don't want to create a Git repository, so leave Source Control unchecked.

Make sure your project will be saved where you want it to be. Click Create.

To get a taste of code completion and the Source Editor as a whole, let's do things a bit differently this time.

1. Open ViewController.swift and under the line import UIKit, begin typing the following (remembering that it's case sensitive):
   import MessageUI

2. As you type the code, Xcode continually suggests the code you are attempting to write so that you can write it quicker.

▪ **Note** You have imported the MessageUI framework because it gives you access to MFMailComposeViewController, among other classes. Go ahead and search for it in Documentation Viewer; you'll find a wealth of information, including confirmation of its parent framework.

3. You need to tell the view controller to act as a delegate for MFMessageComposeViewControllerDelegate and MFMailComposeViewControllerDelegate. To do this, immediately next to class ViewController: UIViewController, type the following, using the code-completion dialog to insert the correct code, as shown in Figure 5-15:

MFMessageComposeViewControllerDelegate, MFMailComposeViewControllerDelegate

Figure 5-15. *The code-completion dialog appears as you add the delegate protocols*

▪ **Note** You can use the up and down arrows to change the selection in the code-completion dialog. Then, with the correct item highlighted, press Enter: your cursor focuses on the end of the line, and the code is entered.

4. When you add the MFMessageComposeViewControllerDelegate protocol, you receive an immediate error; it's important to note that you haven't done anything wrong. The issue is that this protocol has a single delegate function that must be implemented in the class adopting the protocol. This means if you want to add that protocol onto this view controller, the next thing to do is add the delegate function. Before you do that, ensure that the start of your view controller looks like this:

```
import UIKit
import MessageUI
class ViewController: UIViewController, MFMessageComposeViewControllerDelegate,
MFMailComposeViewControllerDelegate {
   override func viewDidLoad() {
      super.viewDidLoad()
      // Do any additional setup after loading the view, typically from a nib.
}
```

5. To remove the error, you can use Xcode's powerful code completion to quickly add the missing delegate method. After the viewDidLoad function, drop down a couple of lines and start typing messageComposeViewController. As you do, code completion kicks in and presents you with the delegate function, as shown in Figure 5-16. Press the Tab key to create the method.

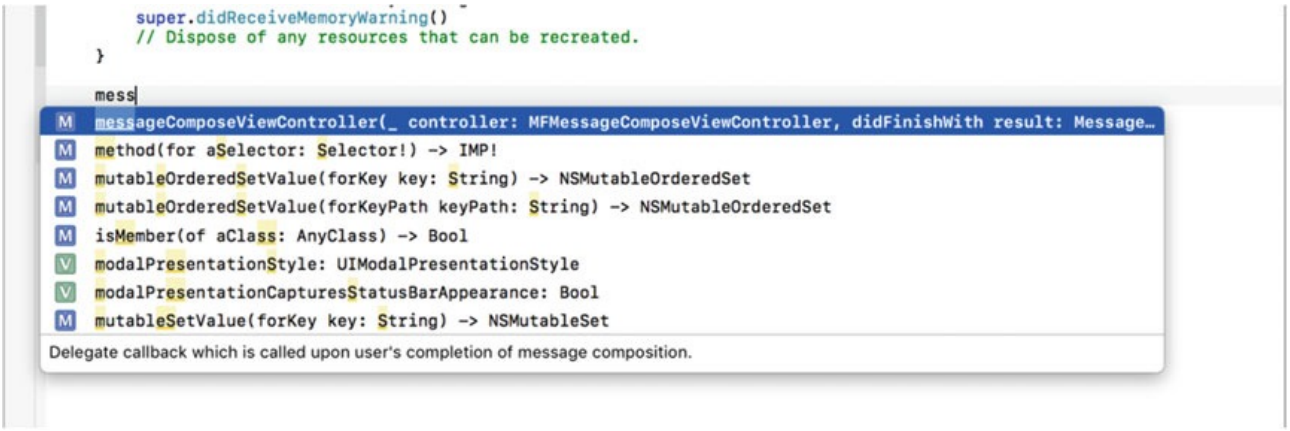


Figure 5-16. *Using code completion to create the missing delegate function*

6. The method was generated with a code placeholder. Remove this with the Backspace key so you're left with an empty function.

7. You are going to manually write three actions that the buttons in your app's interface will use to open the web site or begin composing an e-mail or text message. Begin typing the following highlighted code just before the override func viewDidLoad() line:

```
import UIKit
import MessageUI
class ViewController: UIViewController, MFMessageComposeViewControllerDelegate,
MFMailComposeViewControllerDelegate {
    @IBAction func sendEmail(_ sender: AnyObject) {
    }
    @IBAction func sendText(_ sender: AnyObject) {
    }
    @IBAction func openWebsite(_ sender: AnyObject) {
}
```

Now that you've created the stubs for each action, let's go through the actions and focus on what the code will do before writing it.

Opening Web Sites in Safari

Many applications that you download from the App Store use web views in the native application to load visual information from the Web or from locally stored assets. This is generally frowned on by Apple, which prefers you to write everything natively. In the InTouch app, your goal is to direct users to your company web site, and you do this by forcing Safari to open and display a specified web address. There are some good reasons for using Safari in this instance: first, it's overkill to implement a web view for something that will require a lot of work to create a completely functional implementation with back and forward controls; and, second, if the users open the home page in Safari, they will be able to bookmark it, sync the tab with other iOS devices, and share it on social media.

Locate the openWebsite action stub you just created. Between the braces, begin to type the highlighted code, but feel free to replace http://hse.ru with your own URL:

```
@IBAction func openWebsite(sender: AnyObject) {
UIApplication.shared().open(URL(string: "http://hse.ru")!, options: [:], completionHandler: nil)
}
```

In this code, notice that you aren't creating any variables: you access the UIApplication class and use what are called *type methods* in Swift. If you have experience with other C-based languages, you may be familiar with static methods; a type method is the same thing. In essence, a type method is a function that you can access without instantiating the parent class—that is, without assigning it

to a variable. Type methods are great when you want to quickly access a function without setting a bunch of parameters.

▪ **Note** Each time you add a bracket, notice that for a brief moment a little yellow box appears around its counterpart (that is, the one you're closing). This is to make sure you don't add too many or too few brackets; this also applies to braces.

Sending an E-Mail with MFMailComposeViewController

Next, let's write the code that will allow the users to send an e-mail from your application. What's significant is that you don't need to create an interface for this; you simply use MFMailComposeViewController and preset the values. This is a great approach because unless Apple changes the class, your application will always use the iOS Mail application's compose view, instantly making it familiar to users, more future-proof, and requiring less work than writing your own view.

To implement the view controller, you also have to write another delegate function to handle what happens after the e-mail has been sent. First write the action by adding the following highlighted code:

```
@IBAction func sendEmail(sender: AnyObject) {
   if MFMailComposeViewController.canSendMail()
   {
      let mailVC = MFMailComposeViewController()
      mailVC.setSubject("MySubject")
      mailVC.setToRecipients(["xcode@mattknott.com"])
      mailVC.setMessageBody("<p>Hello!</p>", isHTML: true)
      mailVC.mailComposeDelegate = self;

      self.present(mailVC, animated: true, completion: nil)

   } else {

      print("This device is currently unable to send email")
   }
}
```

Feel free to change xcode@mattknott.com to your own e-mail address, and also feel free to change the subject and presumptuous contents of the e-mail message to whatever you'd like the users to see before they begin to compose their e-mail messages to you.

Next you need to create a new function mailComposeController: didFinishWithResult.

This is a delegate function that is called when the users want to dismiss the mail-compose view controller. In this instance, you account for each of the possible outcomes of trying to send an e-mail before you let the users dismiss the compose view, which they can do after they've sent their

message or if they decide to cancel it. Add the following function before the other delegate function, messageComposeViewController: didFinishWith:

```
func mailComposeController(_ didFinishWithcontroller: MFMailComposeViewController,
    didFinishWith result: MFMailComposeResult, error: NSError?) {
   switch result {
   case MFMailComposeResult.sent:
      print("Result: Email Sent!")
   case MFMailComposeResult.cancelled:
      print("Result: Email Cancelled.")
   case MFMailComposeResult.failed:
      print("Result: Error, Unable to Send Email.")
   case MFMailComposeResult.saved:
      print("Result: Mail Saved as Draft.")
}

   self.dismiss(animated: true, completion: nil)
}
```

You've written all the code needed to send an e-mail. Next you will look at text messaging and how the process is similar to sending an e-mail.

Sending a Text Message

Short Message Service (SMS) messaging is still one of the most popular forms of communication in the world today, and just like e-mail, Apple makes it easy to send a text message from your application. The code is very similar to the previous two methods, but there is one big distinction: you have to test this on a physical device, because the Simulator can't simulate SMS.

With that in mind, in the sendText action, type the following highlighted code:

```
@IBAction func sendText(sender: AnyObject) {

   if MFMessageComposeViewController.canSendText()
   {
      let smsVC : MFMessageComposeViewController = MFMessageComposeViewController()
      smsVC.messageComposeDelegate = self
      smsVC.recipients = ["1234500000"]
      smsVC.body = "Please call me back."
      self.present(smsVC, animated: true, completion: nil)
   } else {

      print("This device is currently unable to send text messages")
   }
}
```

Just as with the e-mail implementation, you now need to complete the code for the delegate function that is called when the process of sending the text message is completed. Once again, you compare the result of the attempt to send a text message against several possible results and print

text to the console based on the outcome. Add the highlighted code to the messageComposeViewController: didFinishWithResult:function:

```
func messageComposeViewController(_ controller: MFMessageComposeViewController,
didFinishWith result: MessageComposeResult) {
   switch result {
   case MessageComposeResult.sent:
      print("Result: Text Message Sent!")
   case MessageComposeResult.cancelled:
      print("Result: Text Message Cancelled.")
   case MessageComposeResult.failed:
      print("Result: Error, Unable to Send Text Message.")
   }
   self.dismiss(animated:true, completion: nil)
}
```

Building the Interface

You've written all the code your application needs to perform three essential communication tasks.

Now you need to build and connect your interface to harness the code you've just written:

1. Open Main.storyboard from the Project Navigator.

2. Drag a label and three buttons onto the view. Position the label at the top of the view and the three buttons beneath it, one on top of the other.

3. Resize the label so it fills the full width of the view, and then open the Attributes Inspector(+ +4).Set the Text attribute to say Ways to get in touch. Center the text and then in the Font attribute, click the T icon to customize the font. Set Font to Custom, Family to Helvetica Neue, Style to Thin, and Size to 23, as shown in Figure 5-17. You may need to increase the height of the label.

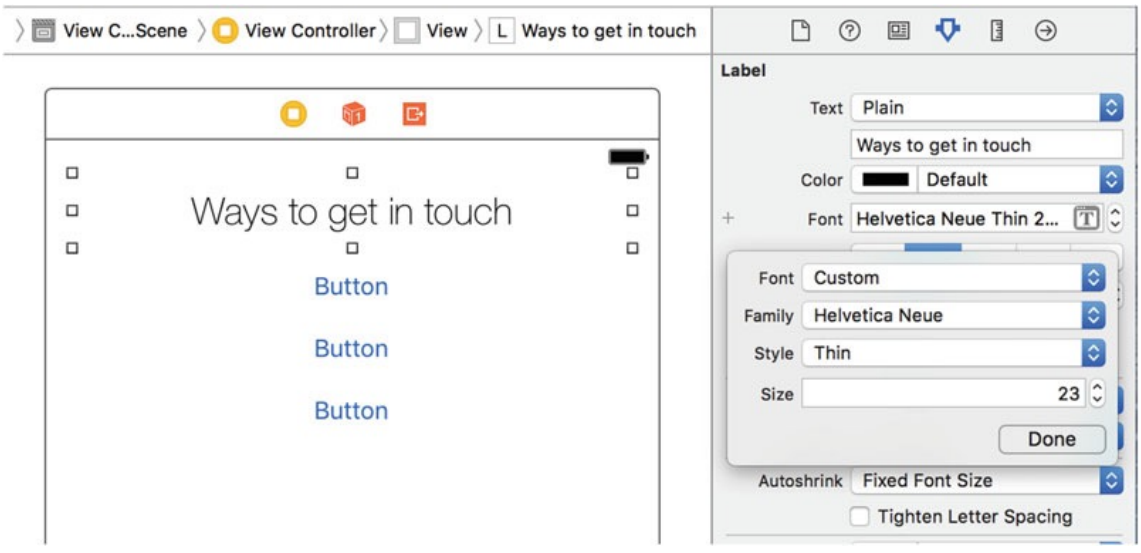


Figure 5-17. *Setting the custom font properties*

4. In order, double-click each of the buttons and name them Email, Text Message, and Website, respectively, before centering them.

5. Use the Resolve Auto Layout Issues button and select Add Missing Constraints under the All Views in View Controller heading to pin the elements in place.

Your finished interface should look something like Figure 5-18.

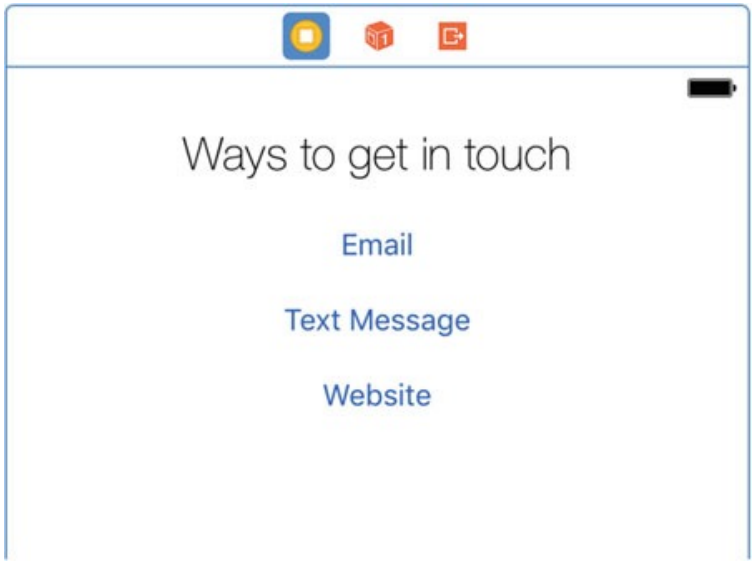


Figure 5-18. *The finished interface*

Making Connections

The code is written and the interface is assembled, but there is no linkage between the two. To address this, you need to connect the actions you've created to the buttons using the Connections Inspector (the sixth and final inspector):

1. Be sure you still have MainStoryboard.storyboard open. If it isn't, open it from the Project Navigator.

2. Open the Connections Inspector (+ +6) with the view controller selected from the document outline, as shown in Figure 5-19. Note that you can select the view controller by clicking the bar at the top in the design area.

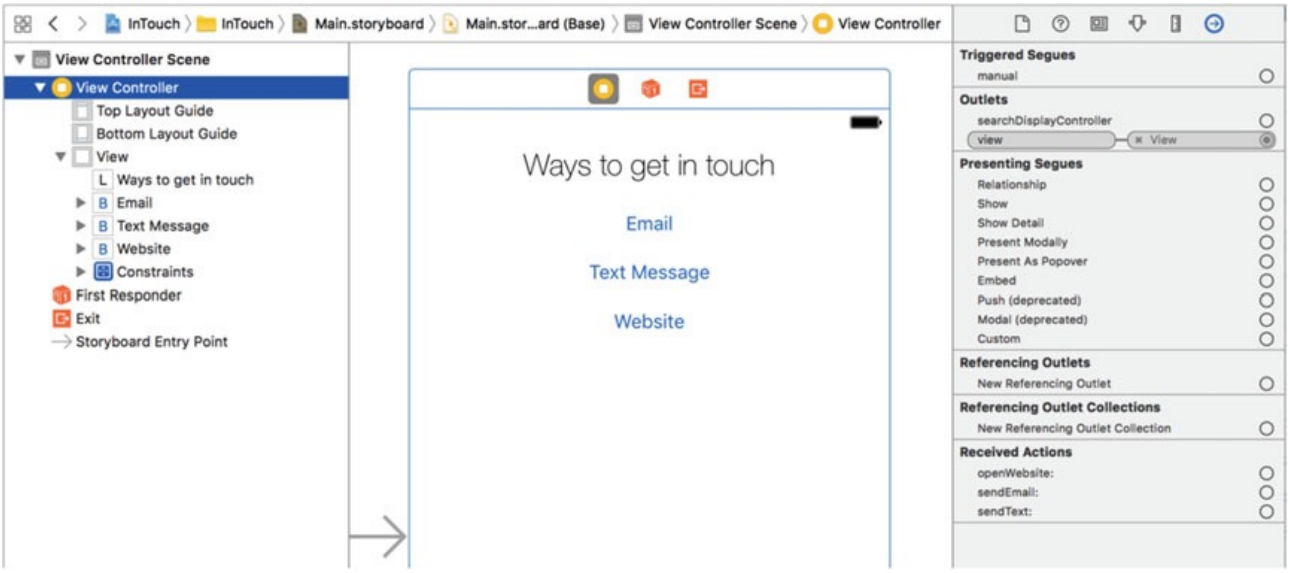


Figure 5-19. *MainStoryboard with the viewcontroller selected and the Connections Inspector open*

3. Under the Received Actions heading in the Connections Inspector, you see the three actions with a hollow circle next to each one. From the sendEmail method's circle, click and drag a connection to the button, as shown in Figure 5-20.

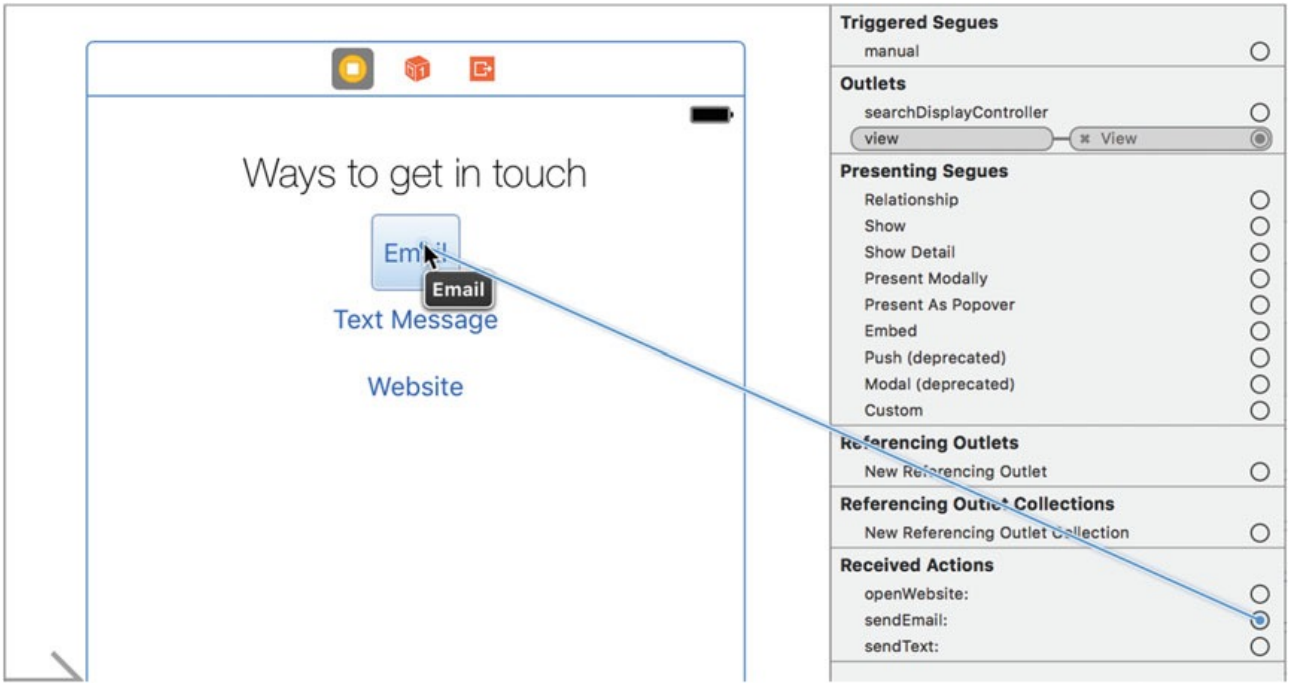


Figure 5-20. *Connecting an action to a button from the Connections Inspector*

4. A menu appears when you release the connection, presenting you with a list of trigger events. The action is called when the correct type of event occurs. Select Touch Up Inside from the bottom part of the list.

▪ **Note** When a button in an iOS application is tapped, the Touch Up Inside event is triggered. By linking the action to this event, you can be sure the code will be executed when the user taps your button.

5. Repeat these steps, linking the two remaining actions to their respective buttons, and making sure to select the Touch Up Inside event from the list.

You've now learned one of several ways to link buttons to preexisting actions! Well done—you're well on your way to becoming an Xcode master.

Running the Application

Run the application on your device or on the iOS Simulator. When you tap the Website button, InTouch is placed in the background and Safari opens. Similarly, if you click the Email or Text Message button, a view is pushed in which the users can send an e-mail or SMS.

■ **Note** You can't send an e-mail from the iOS Simulator, and you can't even see the SMS dialog in it; so in order to fully test this feature, you need to run InTouch on an actual iOS device that has an e-mail account configured.

Summary

This practice explored quite a few different topics, and you added some interesting communication features to your application. Here is what you achieved:

- Used code completion to speed up how you code
- Connected actions using the Connections Inspector
- Added a framework to your project