RAYWENDERLICH

Tutorials for iPhone / iOS Developers and Gamers

How To Secure iOS User Data: The Keychain, Touch ID, and 1Password

Tim Mitra on February 3, 2015

Update 10/07/2015: Updated for Xcode 7 and Swift 2.0

Protecting an app with a login screen is a great way to secure user data – you can use the Keychain, which is built right in to iOS, to ensure that their data stays secure. However, Apple now offers yet another layer of protection with Touch ID, available on the iPhone 5s, 6, 6+, 6S, 6S+ iPad Air 2, iPad Pro, iPad Mini 4 and iPad Mini 3.

And if that weren't enough, with the introduction of extensions in iOS 8, you can even integrate the storage and retrieval of login information using the award-winning 1Password app from AgileBits, thanks to their developers open-sourcing their extension. This means you can comfortably hand over the responsibility of handling login information to the Keychain, TouchID, or 1Password!

In this tutorial you'll start out using the Keychain to store and verify login information. After that, you'll explore Touch ID and then finish off by integrating the 1Password extension into your app.

Learn how to secure your app using Touch ID

Note: Touch ID and 1Password require you test on a physical device,

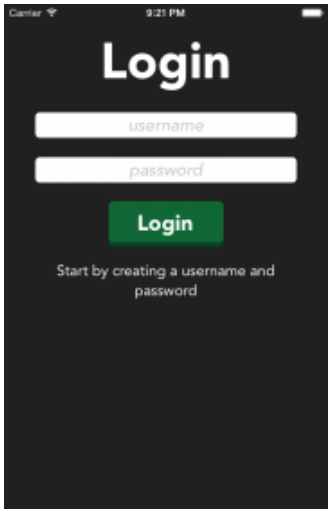
but the Keychain can be used in the simulator

Getting Started

Please download the starter project for this tutorial here.

This is a basic note taking app that uses Core Data to store user notes; the storyboard has a login view where users can enter a username and password, and the rest of the app's views are already connected to each other and ready to use.

Build and run to see what your app looks like in its current state:

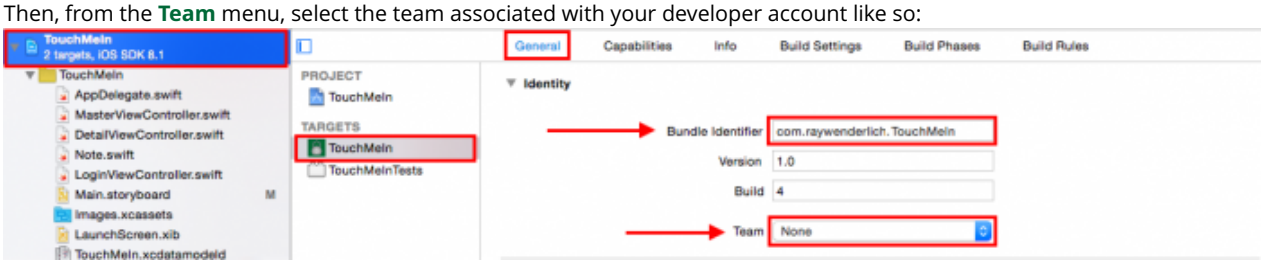


At this point, tapping the **Login** button simply dismisses the view and displays a list of notes – you can also create new notes from this screen. Tapping **Logout** takes you back the login view. If the app is pushed to the background it will immediately return to the login view; this protects data from being viewed without being logged in. This is controlled by setting **Application does not run in background** to **YES** in Info.plist.

Before you do anything else, you should change the Bundle Identifier, and assign an appropriate Team.

Select **TouchMeIn** in the Project Navigator, and then select the **TouchMeIn** target. In the **General** tab change **Bundle Identifier** to use your own domain name, in reverse-domain-notation – for example **com.raywenderich.TouchMeIn**.

Then, from the **Team** menu, select the team associated with your developer account like so:



With all of the housekeeping done, it's time to code! :]

Logging? No. Log In.

To get the ball rolling, you're going to add the ability to check the user-provided credentials against hard-coded values.

Open **LoginViewController.swift** and add the following constants just below where the `managedObjectContext` variable is declared:

```
let usernameKey = "batman"
let passwordKey = "Hello Bruce!"
```

These are simply the hard-coded username and password you'll be checking the user-provided credentials against.

Add the following function below `loginAction(_:)`:

```
func checkLogin(username: String, password: String ) -> Bool {
  if ((username == usernameKey) && (password == passwordKey)) {
    return true
  } else {
    return false
  }
}
```

This checks the user-provided credentials against the constants you defined earlier.

Next, replace the contents of `loginAction(_:)` with the following:

```
if (checkLogin(self.usernameTextField.text!, password: self.passwordTextField.text!)) {
      self.performSegueWithIdentifier("dismissLogin", sender: self)
    }
```

This calls `checkLogin(_:password:)`, which dismisses the login view if the credentials are correct.

Build and run. Enter the username **batman** and the password **Hello Bruce!**, and tap the **Login** button. The login screen should dismiss as expected.

While this simple approach to authentication seems to work, it's not terribly secure, as credentials stored as strings can easily be compromised by curious hackers with the right tools and training. As a best practice, passwords should NEVER be stored directly in the app.

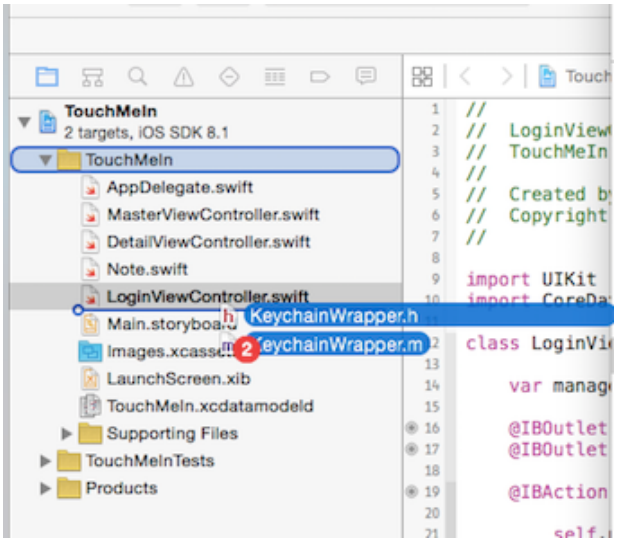
To that end, you'll employ the Keychain to store the password. Check out Chris Lowe's Basic Security in iOS 5 – Part 1 tutorial for the lowdown on how the Keychain works.

The next step is to add a Keychain wrapper class to your app, however it's in Objective-C, so you'll also need to add a bridging header to be able to access the Objective-C class from within Swift.

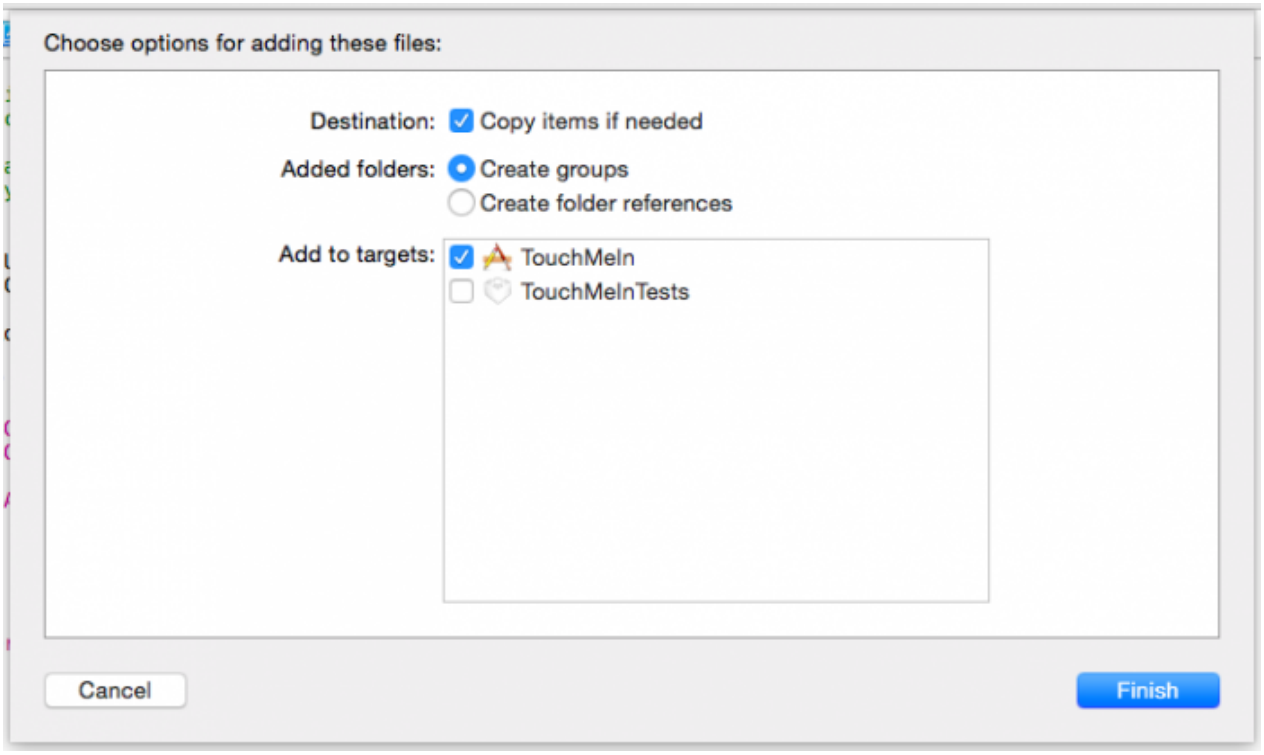
Rapper? No. Wrapper.

You can download **KeychainWrapper** here; this wrapper comes from Apple's Keychain Services Programming Guide.

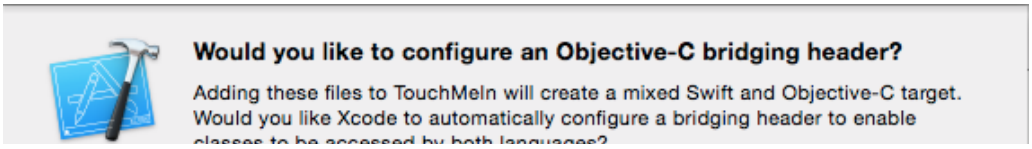
Once downloaded, unzip the archive and drag **KeychainWrapper.h** and **KeychainWrapper.m** into the project, like so:

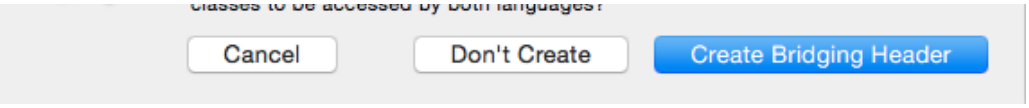


When prompted, make sure that **Copy items if needed** is checked and the **TouchMeIn** target is checked as well:



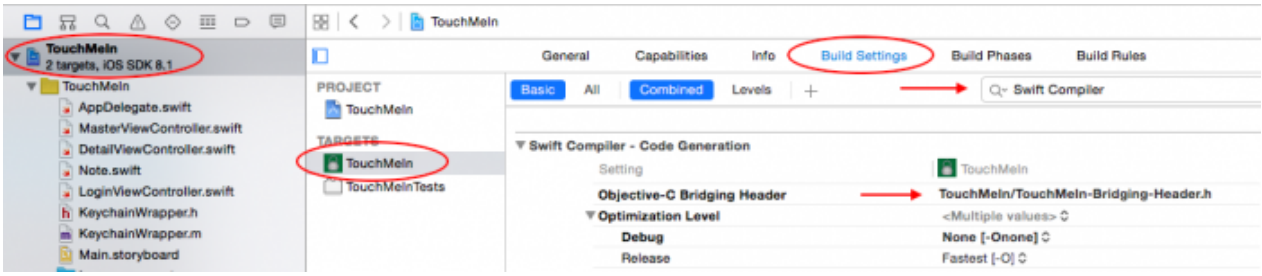
Since you're adding an Objective-C file to a Swift project, Xcode will offer to create the bridging header file – click **Create Bridging Header**:





This creates a bridging header named **TouchMeIn-Bridging-Header.h**. You'll add the Objective-C headers to this file so they can be accessed by your Swift app.

To check that the bridging header is properly setup, select **TouchMeIn** in the Project Navigator. Then navigate to the **Build Settings**. In the search bar, enter **Swift Compiler**, then locate the **Objective-C Bridging Header** field, which should now contain **TouchMeIn/TouchMeIn-Bridging-Header.h**, as shown below:



> **Note**: You can read more about Bridging Headers in Porting Your App to the iPhone 6, iPhone 6 Plus and iOS 8: Top 10 Tips. To learn more about using Swift and Objective-C together, have a look at Apple's Using Swift with Cocoa and Objective-C guide.

Open **TouchMeIn-Bridging-Header.h** and import the Keychain wrapper at the top of the file:

```
#import "KeychainWrapper.h"
```

Build and run to make sure you have no errors. All good? Great — now you can leverage the Keychain from within your app.

> **Note**: If you do see some errors then please refer to Apple's Using Swift with Cocoa and Objective-C guide for help on how to get them resolved.

Keychain, Meet Password. Password, Meet Keychain

To use the Keychain, you first need to store a username and password. After that, you check the user-provided credentials against the Keychain to see if they match.

You'll need to track whether the user has already created some credentials so that you can change the text on the Login button from "Create" to "Login". You'll also store the username in the user defaults so you can perform this check without hitting the Keychain each time.

Open up **LoginViewController.swift** and delete the following lines:

```
let usernameKey = "batman"
let passwordKey = "Hello Bruce!"
```

In their place, add the following:

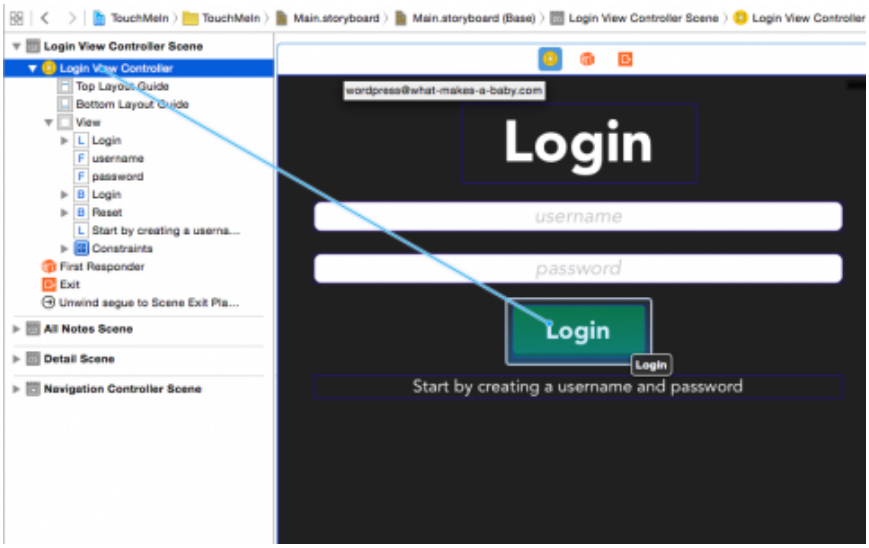
```
let MyKeychainWrapper = KeychainWrapper()
let createLoginButtonTag = 0
let loginButtonTag = 1

@IBOutlet weak var loginButton: UIButton!
```

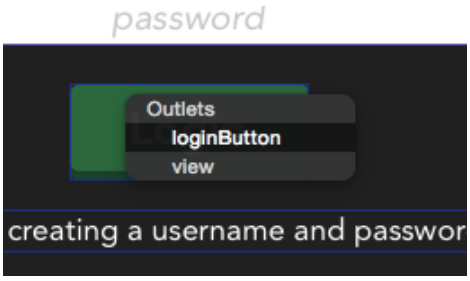
MyKeychainWrapper holds a reference to the Objective-C **KeychainWrapper** class. The next two constants will be used to

determine if the Login button is being used to create some credentials, or to log in; the `loginButton` outlet will be used to update the title of the button depending on that same state.

Open **Main.storyboard** and Ctrl-drag from the **Login View Controller** to the Login button, as shown below:



From the resulting popup, choose **loginButton**:



Next, you need to handle the following two cases for when the button is tapped: if the user hasn't yet created their credentials, the button text will show "Create", otherwise the button will show "Login". You also need to check the entered credentials against the Keychain.

Open **LoginViewController.swift** and replace the code in `loginAction(_:)` with the following:

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

    // 1.
    if (usernameTextField.text == "" || passwordTextField.text == "") {
      let alertView = UIAlertController(title: "Login Problem",
        message: "Wrong username or password." as String, preferredStyle:.Alert)
      let okAction = UIAlertAction(title: "Foiled Again!", style: .Default, handler: nil)
      alertView.addAction(okAction)
      self.presentViewController(alertView, animated: true, completion: nil)
      return;
    }

    // 2.
    usernameTextField.resignFirstResponder()
    passwordTextField.resignFirstResponder()

    // 3.
    if sender.tag == createLoginButtonTag {

      // 4.
      let hasLoginKey = NSUserDefaults.standardUserDefaults().boolForKey("hasLoginKey")
      if hasLoginKey == false {
```

```
        NSUserDefaults.standardUserDefaults().setValue(self.usernameTextField.text, forKey:
"username")
      }

      // 5.
      MyKeychainWrapper.mySetObject(passwordTextField.text, forKey:kSecValueData)
      MyKeychainWrapper.writeToKeychain()
      NSUserDefaults.standardUserDefaults().setBool(true, forKey: "hasLoginKey")
      NSUserDefaults.standardUserDefaults().synchronize()
      loginButton.tag = loginButtonTag

      performSegueWithIdentifier("dismissLogin", sender: self)
    } else if sender.tag == loginButtonTag {
      // 6.
      if checkLogin(usernameTextField.text!, password: passwordTextField.text!) {
        performSegueWithIdentifier("dismissLogin", sender: self)
      } else {
        // 7.
        let alertView = UIAlertController(title: "Login Problem",
          message: "Wrong username or password." as String, preferredStyle:.Alert)
        let okAction = UIAlertAction(title: "Foiled Again!", style: .Default, handler: nil)
        alertView.addAction(okAction)
        self.presentViewController(alertView, animated: true, completion: nil)
      }
    }
}
```

Here's what's happening in the code:

1. If either the username or password is empty, then present an alert to the user and return from the method.
2. Dismiss the keyboard if it's visible.
3. If the login button's tag is **createLoginButtonTag**, then proceed to create a new login.
4. Next, you read **hasLoginKey** from **NSUserDefaults** which indicates whether a password has been saved to the Keychain. If the **username** field is not empty and **hasLoginKey** indicates no login has already been saved, then you save the **username** to **NSUserDefaults**.
5. You then use **mySetObject** and **writeToKeychain** to save the password text to Keychain. You then set **hasLoginKey** in **NSUserDefaults** to **true** to indicate that a password has been saved to the keychain. You set the login button's tag to **loginButtonTag** so that it will prompt the user to log in the next time they run your app, rather than prompting the user to create a login. Finally, you dismiss **loginView**.
6. If the user is logging in (as indicated by **loginButtonTag**), you call **checkLogin(_:password:)** to verify the user-provided credentials; if they match then you dismiss the login view.
7. If the login authentication fails, then present an alert message to the user.

> **Note:** Why not just store the password along with the username in NSUserDefaults? That would be a bad idea because values stored in **NSUserDefaults** are persisted using a plist file. This is essentially an XML file that resides in the app's Library folder, and is therefore readable by anyone with physical access to the device. The Keychain, on the other hand, uses the Triple Digital Encryption Standard (3DES) to encrypt its data.

Next, replace the implementation of **checkLogin(_:password:)** with the following:

```
func checkLogin(username: String, password: String ) -> Bool {
    if password == MyKeychainWrapper.myObjectForKey("v_Data") as? String &&
      username == NSUserDefaults.standardUserDefaults().valueForKey("username") as? String {
        return true
    } else {
      return false
    }
```

```
}
```

This checks that the username entered matches the one stored in **NSUserDefaults** and that the password matches the one stored in the Keychain.

Now you need to set the button title and tags appropriately depending on the state of **hasLoginKey**.

Add the following to **viewDidLoad()**, just below the call to **super**:

```
    // 1.
    let hasLogin = NSUserDefaults.standardUserDefaults().boolForKey("hasLoginKey")

    // 2.
    if hasLogin {
      loginButton.setTitle("Login", forState: UIControlState.Normal)
      loginButton.tag = loginButtonTag
      createInfoLabel.hidden = true
    } else {
      loginButton.setTitle("Create", forState: UIControlState.Normal)
      loginButton.tag = createLoginButtonTag
      createInfoLabel.hidden = false
    }

    // 3.
    if let storedUsername = NSUserDefaults.standardUserDefaults().valueForKey("username") as?
String {
      usernameTextField.text = storedUsername as String
    }
```

Taking each numbered comment in turn:

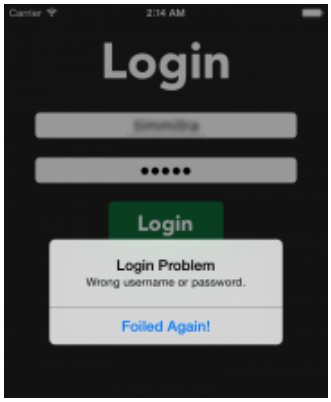
1. You first check **hasLoginKey** to see if you've already stored a login for this user.
2. If so, change the button's title to **Login**, update its tag to **loginButtonTag**, and hide **createInfoLabel**, which contains the informative text "*Start by creating a username and password*". If you don't have a stored login for this user, set the button label to **Create** and display **createInfoLabel** to the user.
3. Finally, you set the username field to what is saved in **NSUserDefaults** to make logging in a little more convenient for the user.

Build and run. Enter a username and password of your own choosing, then tap **Create**.

> **Note:** If you forgot to connect the **loginButton** IBOutlet then you might see the error **Fatal error: unexpectedly found nil while unwrapping an Optional value"**. If you do, connect the outlet as described in the relevant step above.

Now tap **Logout** and attempt to login with the same username and password – you should see the list of notes appear.

Tap **Logout** and try to log in again – this time, use a different password and then tap **Login**. You should see the following alert:



Congratulations – you've now added authentication use the Keychain. Next up, Touch ID.

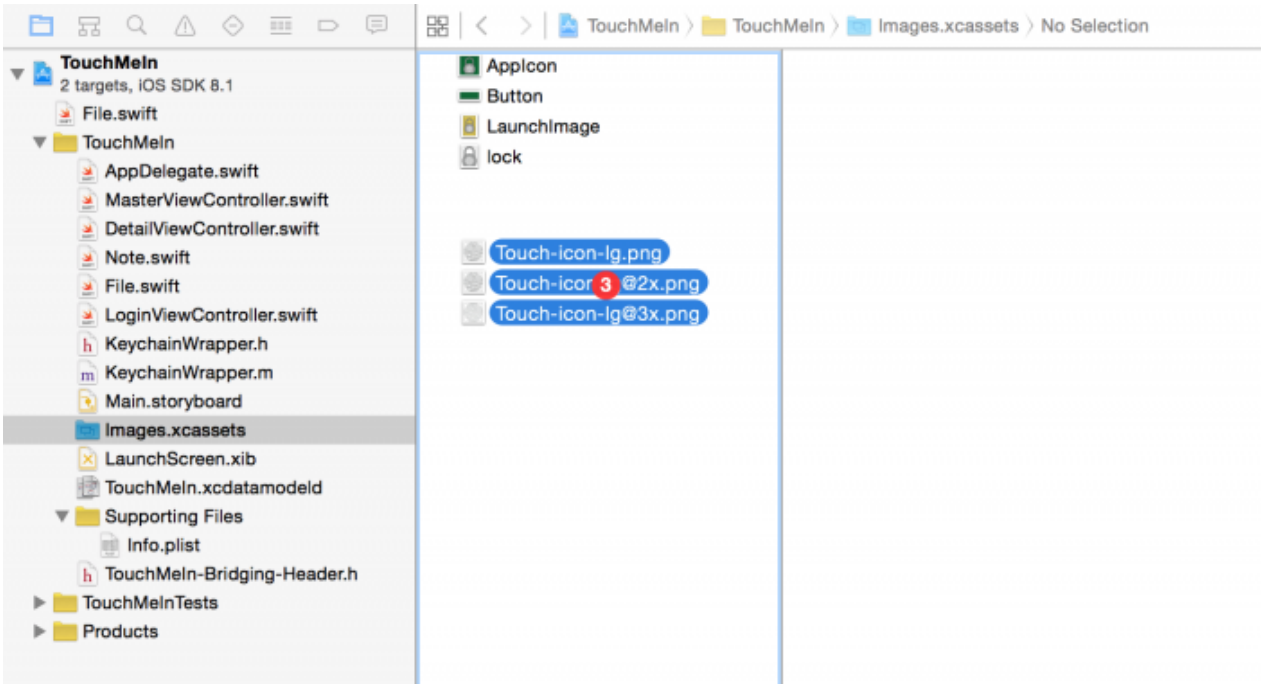
Touching You, Touching Me

> **Note**: In order to test Touch ID, you'll need to run the app on a physical device that supports Touch ID. At the time of this writing, that includes the iPhone 5s, 6, 6+, 6S, 6S+ iPad Air 2, iPad Pro, iPad Mini 4 and iPad Mini 3.

In this section, you'll add Touch ID to your project in addition to using the Keychain. While Keychain isn't necessary for Touch ID to work, it's always a good idea to implement a backup authentication method for instances where Touch ID fails, or for users that don't have a Touch ID compatible device.

Open **Images.xcassets**.

Download the images assets for this part of the project here. Unzip them and open the resulting folder. Locate **Touch-icon-lg.png**, **Touch-icon-lg@2x.png**, and **Touch-icon-lg@3x.png**, select all three and drag them into **Images.xcassets** so that Xcode they're the same image, only with different resolutions:

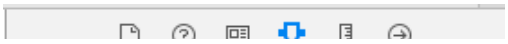


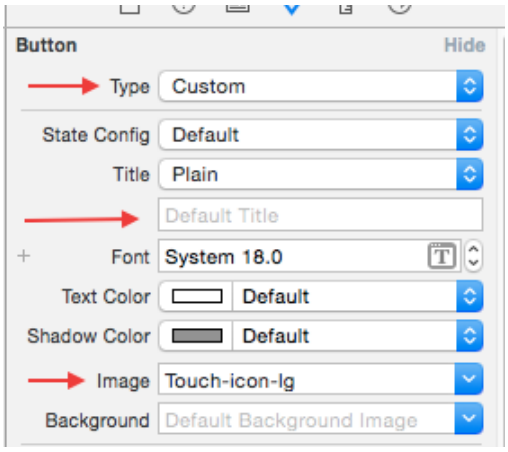
Open up **Main.storyboard** and drag a **Button** from the Object Library onto the **Login View Controller Scene**, just below the label.

Use the Attributes Inspector to adjust the button's attributes as follows:

- Set **Type** to **Custom**.
- Leave the **Title** empty.
- Set **Image** to **Touch-icon-lg**.

When you're done, the button's attributes should look like this:

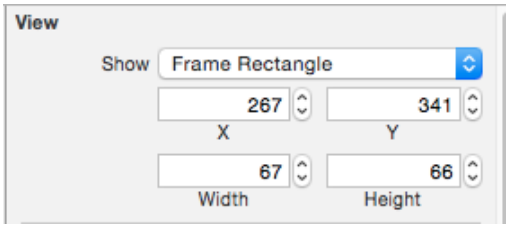




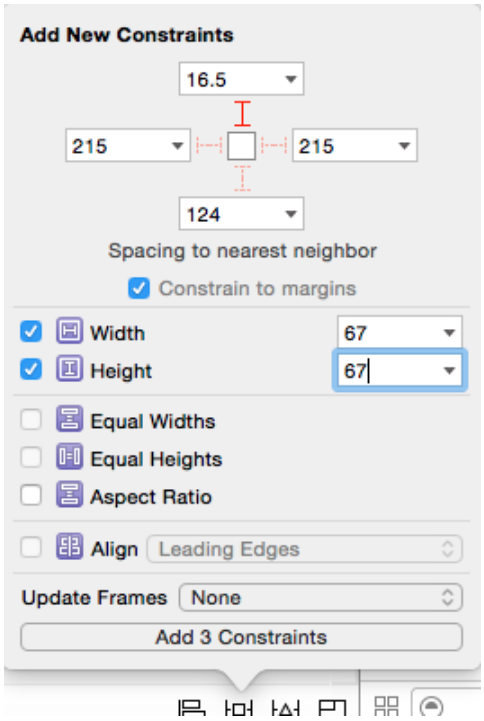
Now adjust your button in the Size Inspector as shown below:

- **Show** should be **Frame Rectangle**
- **X** should be **267**
- **Y** should be **341**
- **Width** should be **67**
- **Height** should be **66**

When you're done, the Size Inspector should look like the following:

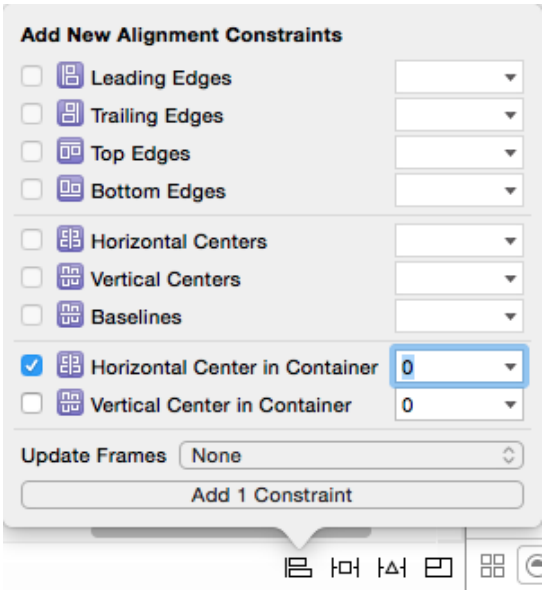


Ensure your new button is selected, then click the **pin** button in the **layout bar** at the foot of the storyboard canvas and set the constraints as below:

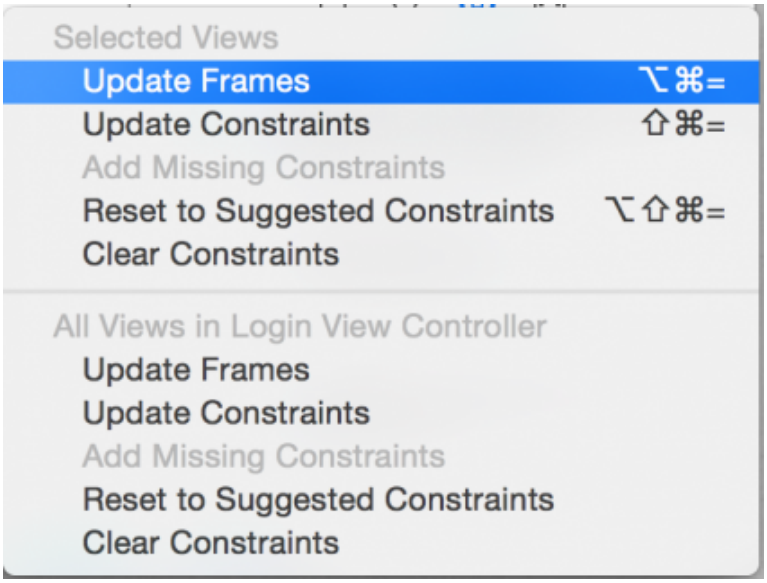


- **Top Space** should be **16.5**
- **Width** should be **67**
- **Height** should be **67**

Next, click the **align** button and check **Horizontal Center in Container**:

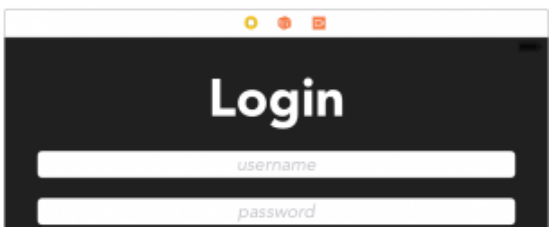


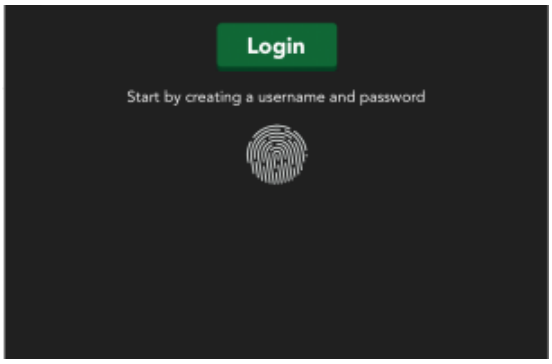
Finally, click the **Resolve Auto Layout Issues** icon and select **Selected Views\Update Frames**, as shown below:



This updates the button's frame to match its new constraints. Tip: it might help to select the button in the Document Outline.

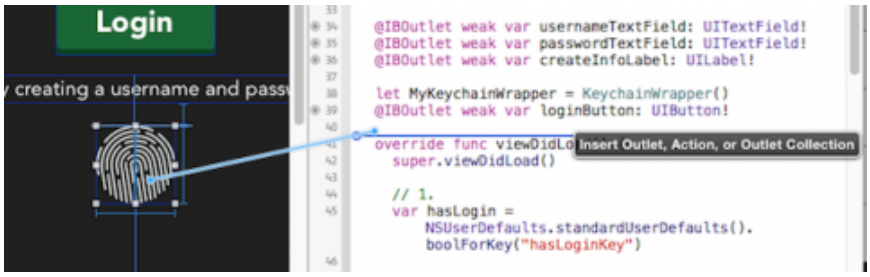
Your view should now look like the following:



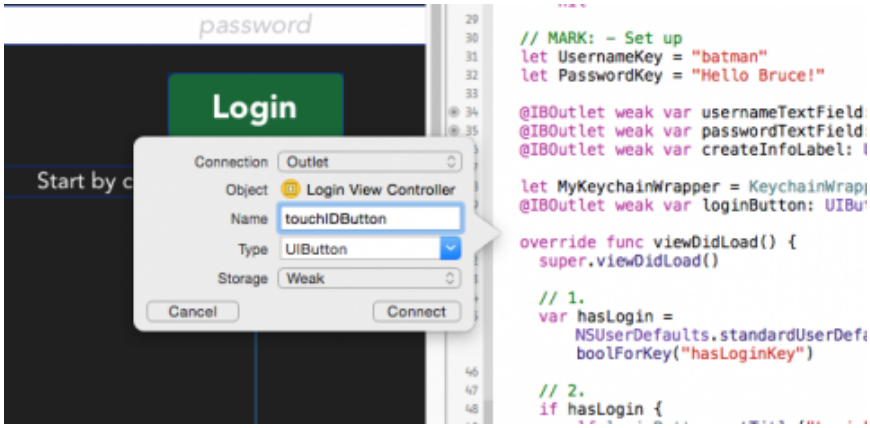


Still in **Main.storyboard**, open the Assistant Editor and make sure **LoginViewController.swift** is showing.

Now, Ctrl-drag from the button you just added to **LoginViewController.swift**, just below the other properties, like so:



In the popup enter **touchIDButton** as the Name and click **Connect**:

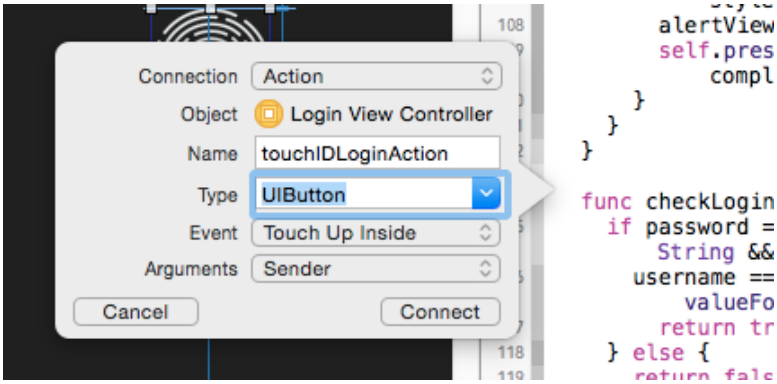


This creates an outlet that you will use to hide the button on devices that don't have Touch ID available.

Now you need to add an action for the button.

Ctrl-drag from the same button to **LoginViewController.swift** to just above **`checkLogin(_:password:)`**:

In the popup, change **Connection** to **Action**, set **Name** to **touchIDLoginAction**, optionally set the **Type** to **UIButton**. Then click **Connect**.



Build and run to check for any errors. You can still build for the Simulator at this point since you haven't yet added support for Touch ID. You'll take care of that now.

Adding Local Authentication

Implementing Touch ID is as simple as importing the **Local Authentication** framework and calling a couple of simple yet powerful methods.

Here's what the documentation the Local Authentication documentation has to say:

"The Local Authentication framework provides facilities for requesting authentication from users with specified security policies."

The specified security policy in this case will be your user's biometrics — A.K.A their fingerprint! :]

Open up **LoginViewController.swift** and add the following import, just below the **CoreData** import:

```
import LocalAuthentication
```

Now you'll need a reference to the **LAContext** class, as well as an **error** property.

Add the following code below the rest of your properties:

```
var context = LAContext()
```

The **context** references an authentication context, which is the main player in Local Authentication.

At the bottom of **viewDidLoad()** add the following:

```
touchIDButton.hidden = true

if context.canEvaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiometrics, error: nil) {
  touchIDButton.hidden = false
}
```

Here you use **canEvaluatePolicy(_:error:)** to check whether the device can implement Touch ID authentication. If so, then show the Touch ID button; if not, then leave it hidden.

Build and run on the Simulator; you'll see the Touch ID logo is hidden. Now build and run on your physical Touch ID-capable device; you'll see the Touch ID button is displayed.

Putting Touch ID to Work

Still working in **LoginViewController.swift**, replace the entire **touchIDLoginAction()** method with the following:

```
  // MARK: - Login with TouchID

  @IBAction func touchIDLoginAction() {
    // 1.
    if context.canEvaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiometrics, error:nil)
{
```

```
    // 2.
    context.evaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiometrics,
      localizedReason: "Logging in with Touch ID",
      reply: { (success : Bool, error : NSError? ) -> Void in

        // 3.
        dispatch_async(dispatch_get_main_queue(), {
          if success {
            self.performSegueWithIdentifier("dismissLogin", sender: self)
          }

          if error != nil {

            var message : NSString
            var showAlert : Bool

            // 4.
            switch(error!.code) {
            case LAError.AuthenticationFailed.rawValue:
              message = "There was a problem verifying your identity."
              showAlert = true
              break;
            case LAError.UserCancel.rawValue:
              message = "You pressed cancel."
              showAlert = true
              break;
            case LAError.UserFallback.rawValue:
              message = "You pressed password."
              showAlert = true
              break;
            default:
              showAlert = true
              message = "Touch ID may not be configured"
              break;
            }

            let alertView = UIAlertController(title: "Error",
              message: message as String, preferredStyle:.Alert)
            let okAction = UIAlertAction(title: "Darn!", style: .Default, handler: nil)
            alertView.addAction(okAction)
            if showAlert {
              self.presentViewController(alertView, animated: true, completion: nil)
            }

          }
        })

    })
  } else {
    // 5.
    let alertView = UIAlertController(title: "Error",
      message: "Touch ID not available" as String, preferredStyle:.Alert)
    let okAction = UIAlertAction(title: "Darn!", style: .Default, handler: nil)
    alertView.addAction(okAction)
    self.presentViewController(alertView, animated: true, completion: nil)

  }

}
```

Here's what's going on in the code above:

1. Once again, you're using `canEvaluatePolicy(_:error:)` to check whether the device is Touch ID capable.
2. If the device does support Touch ID, you then use `evaluatePolicy(_:localizedReason:reply:)` to begin the policy evaluation — that is, prompt the user for Touch ID authentication. `evaluatePolicy(_:localizedReason:re-`

`ply:)` takes a reply block that is executed after the evaluation completes.

3. Inside the reply block, you handling the success case first. By default, the policy evaluation happens on a private thread, so your code jumps back to the main thread so it can update the UI. If the authentication was successful, you call the segue that dismisses the login view.
4. Now for the "failure" cases. You use a `switch` statement to set appropriate error messages for each error case, then present the user with an alert view.
5. If `canEvaluatePolicy(_:error:)` failed, you display a generic alert. In practice, you should really evaluate and address the specific error code returned, which could include any of the following:
   - `LAErrorTouchIDNotAvailable`: the device isn't Touch ID-compatible.
   - `LAErrorPasscodeNotSet`: there is no passcode enabled as required for Touch ID
   - `LAErrorTouchIDNotEnrolled`: there are no fingerprints stored.

iOS responds to `LAErrorPasscodeNotSet` and `LAErrorTouchIDNotEnrolled` on its own with relevant alerts.

Build and run on a physical device and test logging in with Touch ID.

Since `LAContext` handles most of the heavy lifting, it turned out to be relatively straight forward to implement Touch ID. As a bonus, you were able to have Keychain and Touch ID authentication in the same app, to handle the event that your user doesn't have a Touch ID-enabled device.

In the third and final part of this tutorial, you'll use 1Password's iOS 8 extension to store and retrieve login information, along with other sensitive user data.

1Password To Rule Them All

The password manager 1Password from Agilebits runs on iOS, OS X, Windows, and Android. It stores login credentials, software licenses and other sensitive information in a virtual vault locked with a PBKDF2-encrypted master password. In this section, you'll learn how to store user credentials in 1Password via the extension, and later retrieve them in order to authenticate a user.

> **Note**: You'll need 1Password installed on a physical device in order to follow along with the next section.

To begin with, you need to add some new graphics that will be used with a 1Password-specific button.

Open **Images.xcassets**. Then, amongst the assets you downloaded earlier, locate the following three files:

- **onepassword-button.png**
- **onepassword-button@2x.png**
- **onepassword-button@3x.png**

Select them all and drag them as one into **Images.xcassets**. Then, find the three files:

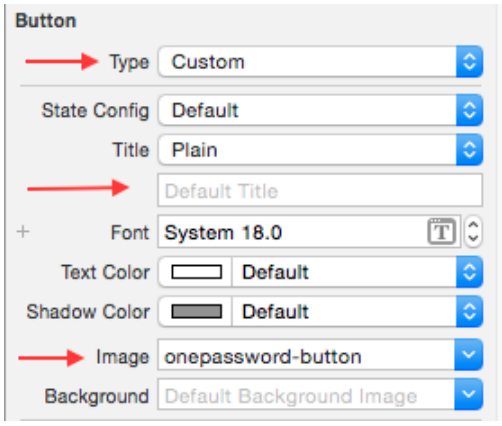
- **onepassword-button-green.png**
- **onepassword-button-green@2x.png**
- **onepassword-button-green@3x.png**

And again, select them all and drag them as one into **Images.xcassets**.

Open **Main.storyboard** and drag a **Button** from the Object Library on to the **Login View Controller Scene**, just below the Touch ID button. Using the Attributes Inspector, adjust the button's attributes as follows:

- Set **Type** to **Custom**
- Leave **Title** empty
- Set **Image** to **onepassword-button**

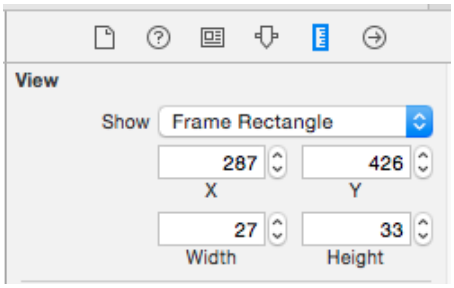
The Attributes Inspector should now look like the following:



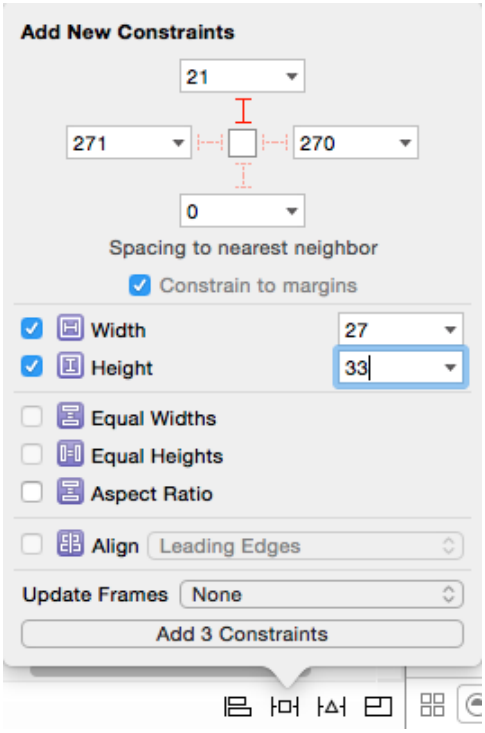
Using the Size Inspector, adjust the placement and size as per the following:

- Set **X** to **287**
- Set **Y** to **426**
- Set **Width** to **27**
- Set **Height** to **33**

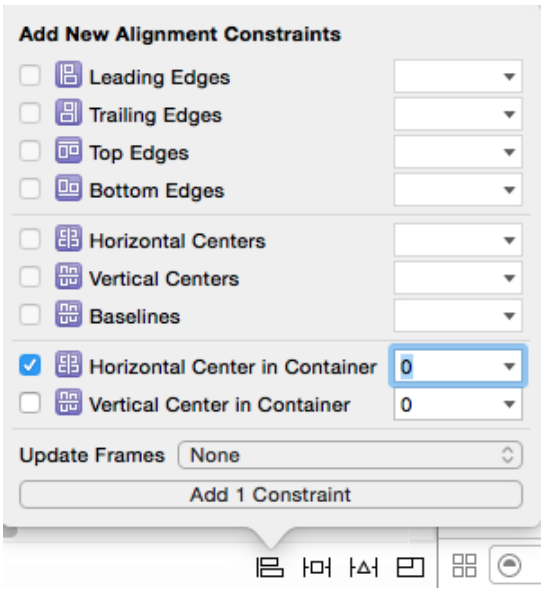
You can check your size and placement against the image below:



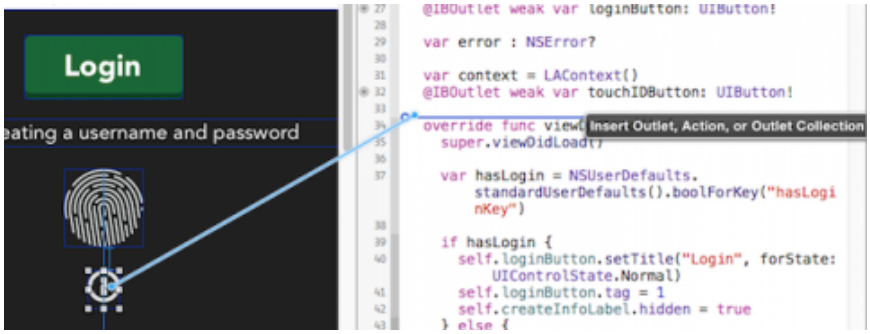
With the button still selected, click **pin** in the layout bar at the bottom of the storyboard canvas and set the button's **Top Space** to **21**, **Width** to **27**, and **Height** to **33**:



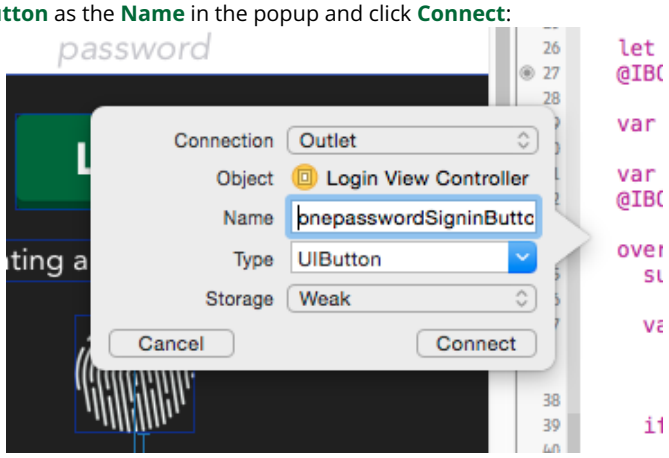
Next, click **align** in the layout bar and check **Horizontal Center in Container**:



Still working with **Main.storyboard**, open the Assistant Editor, which itself should open up **LoginViewController.swift** – if it doesn't, select the file from the jump-bar. Now **Ctrl-drag** from the button to **LoginViewController.swift**, just below the other properties, as shown below:



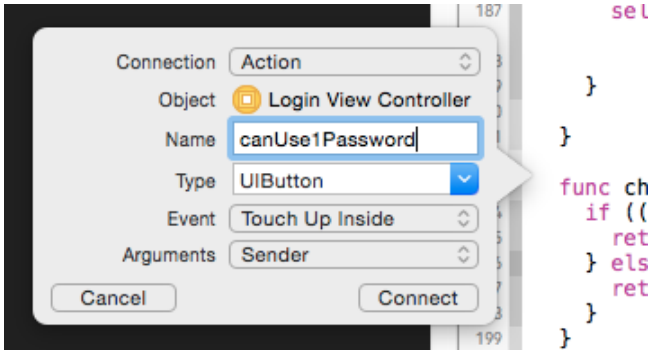
Enter **onepasswordSigninButton** as the **Name** in the popup and click **Connect**:



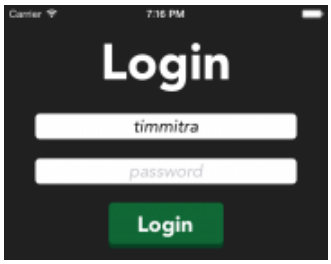
This creates an `IBOutlet` that you will use to change the 1Password button image depending on whether it's available or not.

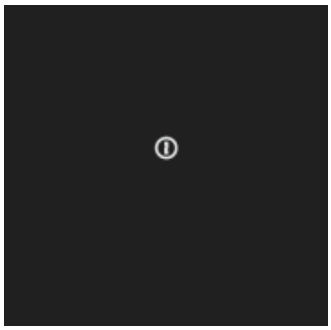
Next, add an action for `onepasswordSigninButton`. **Ctrl-drag** from the button to **LoginViewController.swift**, just above `checkLogin(_:password:)`:

Change the **Connection** type to **Action**. Set the **Name** to **canUse1Password** and leave **Arguments** set to **Sender** and click **Connect**.



Build and run. You should see your new 1Password button displayed, as shown below:

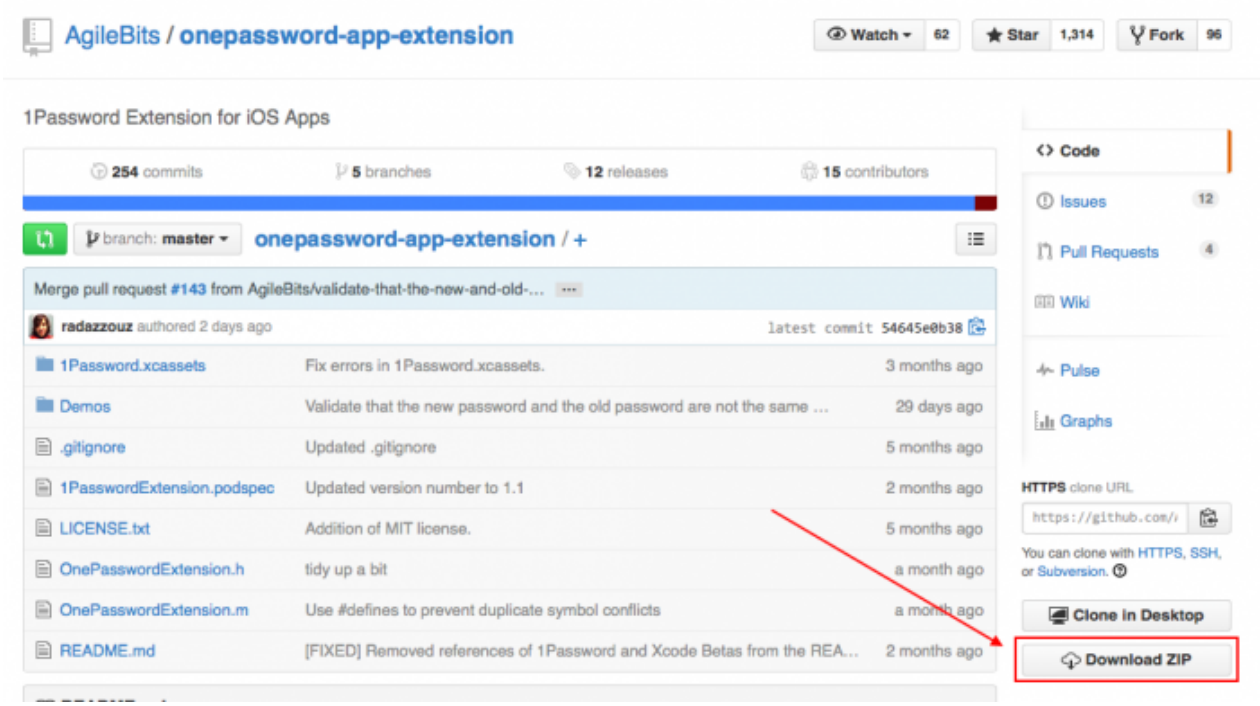




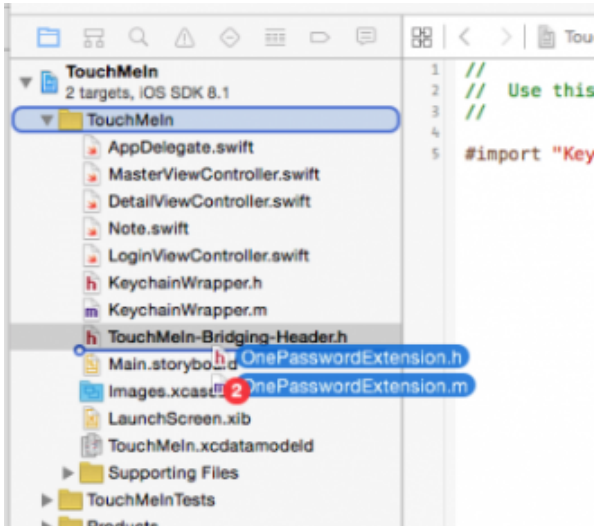
An Agile Extension

Now that the interface is laid out, you can start implementing the 1Password support.

Head to the 1Password Extension repository on GitHub and click **Download ZIP**, as indicated below:



Unzip the downloaded file. Open the folder and drag the **OnePasswordExtension.h** and **OnePasswordExtension.m** files into your project, like so:



Make sure that **Copy items if needed** and the **TouchMeIn** target are both checked.

Recall that when you add Objective-C files to a Swift project, Xcode offers to make a bridging header. Since you've already created a bridging header in an earlier step you can simply add the 1Password header to that.

Open **TouchMeIn-Bridging-Header.h** and add the following import:

```
#import "OnePasswordExtension.h"
```

Open **LoginViewController.swift** and add the following import to the top of the file:

```
import Security
```

This simply imports the Security framework, which is required by the 1Password extension.

Now add the following properties to the top of **LoginViewController**:

```
let MyOnePassword = OnePasswordExtension()
var has1PasswordLogin: Bool = false
```

The former holds a reference to the main class of the 1Password extension, and the latter tracks whether a 1Password record has already been created; you set it to default as **false** as there won't be one on first run.

Next update **viewDidLoad()** by replacing the entire **if hasLogin** block with the following:

```
if hasLogin {
  loginButton.setTitle("Login", forState: UIControlState.Normal)
  loginButton.tag = loginButtonTag
  createInfoLabel.hidden = true
  onepasswordSigninButton.enabled = true
} else {
  loginButton.setTitle("Create", forState: UIControlState.Normal)
  loginButton.tag = createLoginButtonTag
  createInfoLabel.hidden = false
  onepasswordSigninButton.enabled = false
}
```

This disables the **onepasswordSigninButton** until the initial username and password have been stored in the Keychain.

Now you'll need to access the 1Password Extension and test whether the iOS app is installed and available, and enable the 1Password button if it is.

Add the following to **viewDidLoad()**:

```
onepasswordSigninButton.hidden = true
let has1Password = NSUserDefaults.standardUserDefaults().boolForKey("has1PassLogin")

if MyOnePassword.isAppExtensionAvailable() {
  onepasswordSigninButton.hidden = false
  if has1Password {
    onepasswordSigninButton.setImage(UIImage(named: "onepassword-button") , forState: .Normal)
  } else {
    onepasswordSigninButton.setImage(UIImage(named: "onepassword-button-green") , forState:
.Normal)
  }
}
```

This hides **onepasswordSigninButton** by default, and then shows it only if the extension is installed. You then set **has1Password** from the **has1PassLogin** key stored in **NSUserDefaults** to indicate whether a 1Password record has already been created, and set the button image accordingly.

Next, modify **canUse1Password** to output a simple message to the console when you tap the 1Password button:

```
  @IBAction func canUse1Password(sender: UIButton) {
    print("one password")
  }
```

Build and run your app on both the simulator and your physical device with 1Password installed. The 1Password button should be hidden on the simulator, and it should show in green on your physical device. Tap the 1Password button and the following should print to the console:

```
one password
```

Wait a minute! The button doesn't appear on the device. On iOS 9 we are now required to add a **Custom URL Scheme** because of the Privacy settings. You can find more info in the WWDC 2015 video Privacy and Your Apps.

Open your app's **Info.plist** and add the following to the bottom. In the last row, tap the **+** to add a new row. Enter **LSApplicationQueriesSchemes** and set it to **Array**. Click the disclosure triangle and then add a new **Item0**, set it to String and enter **org-appextension-feature-password-management**.

LSApplicationQueriesSchemes	Array	(1 item)
Item 0	String	org-appextension-feature-password-management

Build and Run on your device and the button should now be visible.

Tapping into the Power of 1Password

There are a few methods you'll use in the 1Password extension: **`storeLoginForURLString(_:loginDetails:passwordGenerationOptions:forViewController:sender:)`** lets you create some login credentials, while **`findLoginForURLString(_:forViewController:sender:completion:)`** retrieves the credentials from the 1Password vault.

Open **LoginViewController.swift**, and add the following new method to it:

```
func saveLoginTo1Password(sender: AnyObject) {
    // 1.
    let newLoginDetails : NSDictionary = [
      AppExtensionTitleKey: "Touch Me In",
      AppExtensionUsernameKey: self.usernameTextField.text!,
      AppExtensionPasswordKey: self.passwordTextField.text!,
      AppExtensionNotesKey: "Saved with the TouchMeIn app",
      AppExtensionSectionTitleKey: "Touch Me In app",
    ]

    // 2.
    let passwordGenerationOptions : NSDictionary = [
      AppExtensionGeneratedPasswordMinLengthKey: 6,
      AppExtensionGeneratedPasswordMaxLengthKey: 50
    ]

    // 3.
    MyOnePassword.storeLoginForURLString("TouchMeIn.Login",
      loginDetails: newLoginDetails as! [String : String],
      passwordGenerationOptions: passwordGenerationOptions as [NSObject : AnyObject],
      forViewController: self, sender: sender) { (loginDict : [NSObject : AnyObject]!,
      error : NSError!) -> Void in

      // 4.
      if loginDict == nil {
        if error.code != AppExtensionErrorCodeCancelledByUser {
          print("Error invoking 1Password App Extension for find login: \(error)")
        }
        return
      }

      // 5.
      let foundUsername = loginDict["username"] as! String
      let foundPassword = loginDict["password"] as! String

      // 6.
      if self.checkLogin(foundUsername, password: foundPassword) {

        self.performSegueWithIdentifier("dismissLogin", sender: self)
```

```
      } else {

        // 7.
        let alertView = UIAlertController(title: "Error", message: "The info in 1Password is
incorrect" as String, preferredStyle:.Alert)
        let okAction = UIAlertAction(title: "Darn!", style: .Default, handler: nil)
        alertView.addAction(okAction)
        self.presentViewController(alertView, animated: true, completion: nil)

      }
    // TODO - add NSUserDefaults check
  }
}
```

That's a fair bit of code; here's what's happening in detail:

1. You create a dictionary with the username and password provided by the user, as well as the keys required by the 1Password extension. This dictionary will be passed to the extension when saving the password.
2. You add some optional parameters which are used by 1Password to generate a password; here you're simply stating the minimum and maximum lengths of the password. The extension will provide its own defaults for these values if you don't configure them here.
3. You provide a string — `TouchMeIn.Login` — to identify the saved record; you also pass in `newLoginDetails` and `passwordGenerationOptions`, the two dictionaries you created in steps 1 and 2 respectively.
4. If all goes well in step 3, you'll receive a `loginDict` in return, which contains the username and password; if not, you'll print an error to the console and return.
5. You extract the username and password from `loginDict`.
6. You call `checkLogin(_:password:)` with the username and password you obtained in the previous step; if the credentials match the information stored in the Keychain, then you log the user in and dismiss the login view.
7. If the login information is incorrect, you display an alert with an error message.

> **Note:** 1Password uses a string – `URLString` – as the key of the record in the vault. You can use any unique string, and although it's tempting to use your Bundle ID, you should really create a unique human-friendly string for this purpose instead as the string will be visible to the user in the 1Password app. In your case, you use the string `TouchMeIn.Login`

Now you need to check whether the username is stored in the NSUserDefaults. If not, then you'll be defensive in your coding and set the value from the text field.

Find the following line in `saveLoginTo1Password(_:)`:

```
// TODO - add NSUserDefaults check
```

…and replace it with the following:

```
if NSUserDefaults.standardUserDefaults().objectForKey("username") != nil {
  NSUserDefaults.standardUserDefaults().setValue(self.usernameTextField.text, forKey: "user-
name")
}
```

This sets the `username` key in the user defaults to the value of the username text field if it's not already set.

Add the following code to the bottom of `saveLoginTo1Password(_:)`:

```
NSUserDefaults.standardUserDefaults().setBool(true, forKey: "has1PassLogin")
NSUserDefaults.standardUserDefaults().synchronize()
```

This updates `has1PassLogin` to indicate that the user created a 1Password record. It's a simple way to prevent the creation of a second entry in the vault. You'll check `has1PassLogin` later on as well before attempting to save credentials to 1Password.

Now you need a way to look up the stored 1Password login information for your app.

Add the following method just above **checkLogin(_:password:)**:

```
// MARK: - Login with 1Password
@IBAction func findLoginFrom1Password(sender: AnyObject) {

  MyOnePassword.findLoginForURLString( "TouchMeIn.Login",
    forViewController: self,
    sender: sender,
    completion: { (loginDict : [NSObject: AnyObject]!, error : NSError!) -> Void in

      // 1.
      if loginDict == nil {
        if error.code != AppExtensionErrorCodeCancelledByUser {
          print("Error invoking 1Password App Extension for find login: \(error)")
        }
        return
      }

      // 2.
      if NSUserDefaults.standardUserDefaults().objectForKey("username") == nil {
        NSUserDefaults.standardUserDefaults().setValue(loginDict[AppExtensionUsernameKey],
          forKey: "username")
        NSUserDefaults.standardUserDefaults().synchronize()
      }

      // 3.
      let foundUsername = loginDict["username"] as! String
      let foundPassword = loginDict["password"] as! String

      if self.checkLogin(foundUsername, password: foundPassword) {

        self.performSegueWithIdentifier("dismissLogin", sender: self)

      } else {

        let alertView = UIAlertController(title: "Error",
          message: "The info in 1Password is incorrect" as String, preferredStyle:.Alert)
        let okAction = UIAlertAction(title: "Darn!", style: .Default, handler: nil)
        alertView.addAction(okAction)
        self.presentViewController(alertView, animated: true, completion: nil)
      }

  })
}
```

This function uses the 1Password method **findLoginForURLString(_:forViewController:sender:completion:)** to look up the vault record for the passed-in **URLString**, which in this case is **TouchMeIn.Login**; it then executes a completion block on success which has access to the returned dictionary.

Here's a closer look at the code:

1. If the **loginDict** is **nil** something has gone wrong, so you print an error message to the console based on the provided **NSError** and return.
2. Here you check if the username is already stored in NSUserDefaults. If it isn't, you add it from the results returned by 1Password.
3. This passes the retrieved values into **checkLogin(_:password:)**. If the login matches the one stored in Keychain, you dismiss the login controller. However, if there's a mismatch, then you display an alert to the user.

Now you'll finish implementing **canUse1Password(_:)** to call either **saveLoginTo1Password(_:)** on first run, or **findLoginFrom1Password(_:)** if **has1PassLogin** indicates there's stored login information for this user.

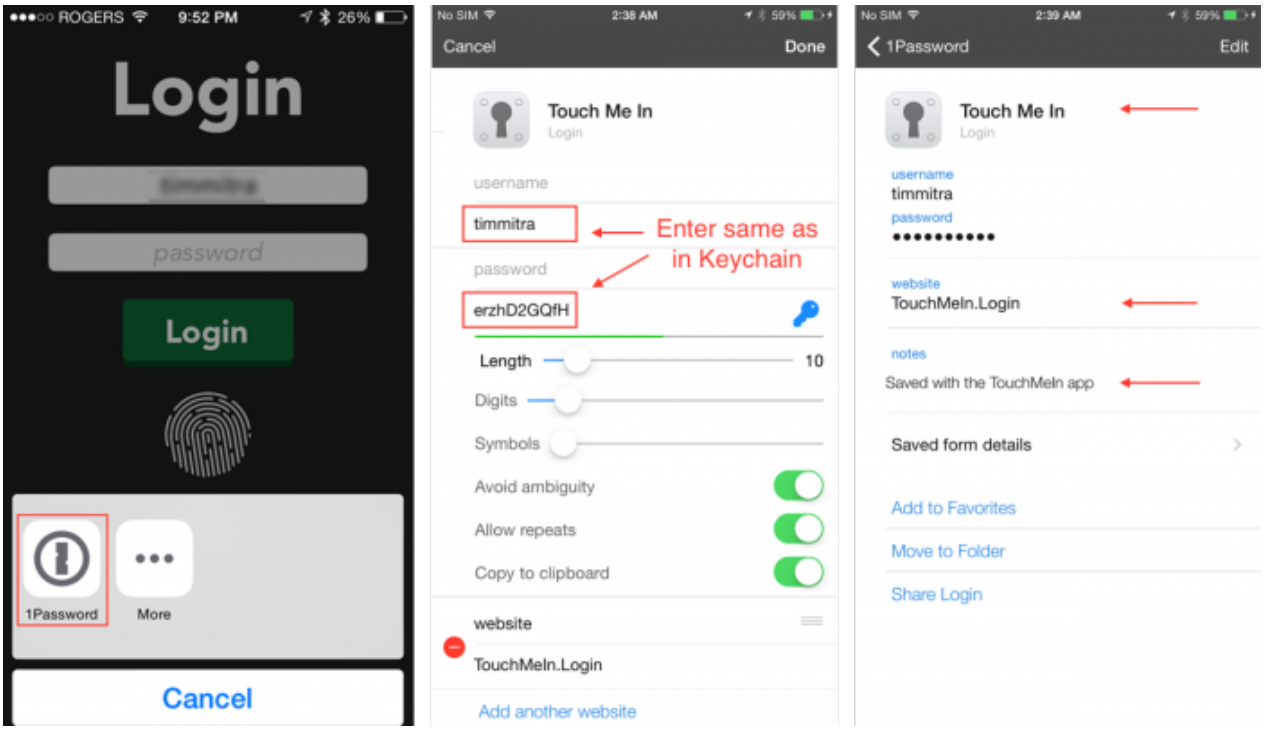
Replace the existing **canUse1Password** implementation with the following:

```
@IBAction func canUse1Password(sender: AnyObject) {
```

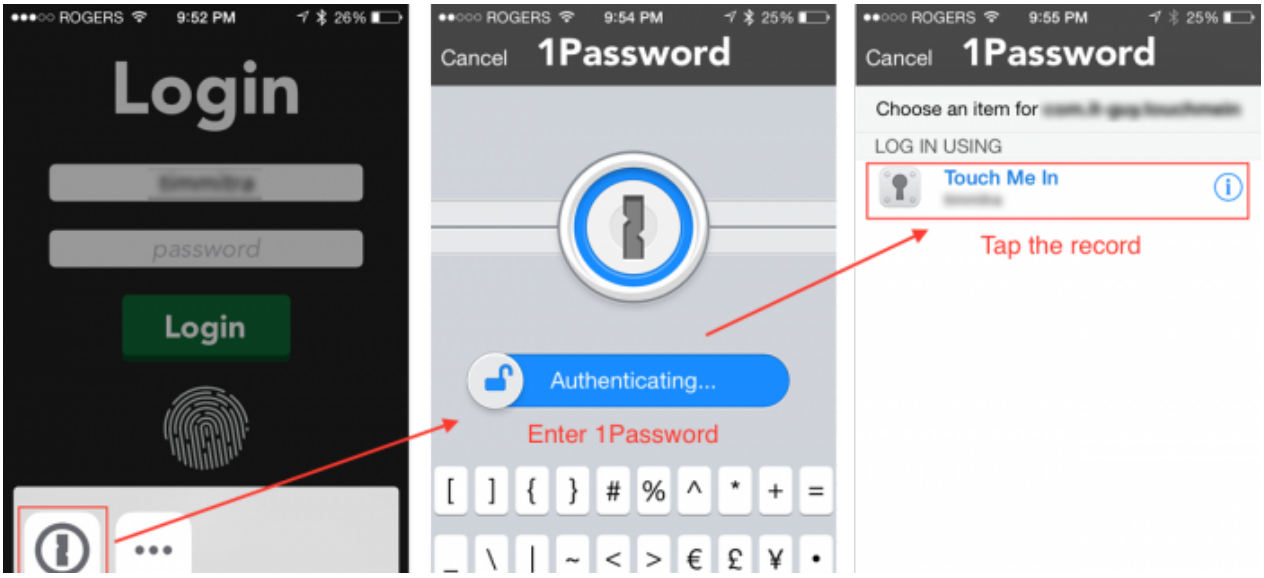
```
  if NSUserDefaults.standardUserDefaults().objectForKey("has1PassLogin") != nil {
    self.findLoginFrom1Password(self)
  } else {
    self.saveLoginTo1Password(self)
  }
}
```

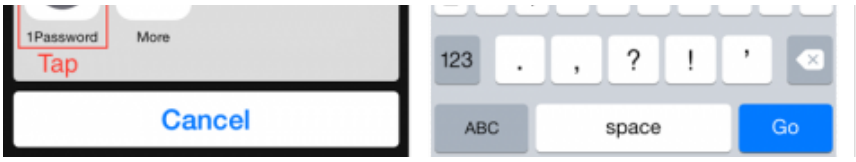
Build and run. Tap the 1Password button to store your login credentials. A 1Password icon will appear in the Action sheet – tap it to launch the extension. The extension's view is presented modally and you can login with your system password, or Touch ID if you're using your physical device. 1Password then presents the record for your app, using the identifier you set up earlier.

You'll see screens similar to those shown below. By default 1Password will generate a password. For this test, be sure to enter the same credentials you used to set up your login in the Keychain in earlier runs. Once that's complete, hit the **Done** button. The last screenshot shows what you'll see if you later edit the record – which shows that it has the vault name, URL and notes you provided when creating the record.



Now that you've stored the login, subsequent taps of the 1Password button will retrieve the vault credentials and log you in if the credentials match the Keychain. Try to log in again and you should see the screens as shown below:





That's it — you've now implemented support for the 1Password extension – a powerful way to improve your user login experience for 1Password users!

Where to Go from Here?

You can download the completed sample application from this tutorial here.

The `LoginViewController` you've created in this tutorial provides a jumping-off point for any app that needs to manage user credentials.

You can also add a new view controller, or modify the existing `LoginViewController`, to allow the user to change their password from time to time. This isn't necessary with Touch ID, since the user's biometrics probably won't change much in their lifetime! :] However, you could create a way to update the Keychain and then update 1Password accordingly; you'd want to prompt the user for their current password before accepting their modification.

As always, if you have any questions or comments on this tutorial, feel free to join the discussion below!

Tim Mitra

Tim is a mobile app developer and artist. He also teaches iOS dev, Swift and Objective-C. Tim also does web development and he runs iT Guy Technologies, a software development company in Toronto, Canada. He studied Fine Arts before there were Macs and he's worked for many years in software development, IT, graphic design, publishing and printing. He is also the founder and host of the MTJC Podcast on app development & business