# Basic Exercises Part 11.3 Local Authentication

## Touch ID

* TouchID is the biometric authentication technology that Apple introduced in iOS 7 and used in iPhone 5s to allow users to unlock their devices and make purchases in the App Store.
* Since iOS 8 things have changed, and TouchID has being used as an authentication mechanism.
* The usage of the TouchID is based on a new framework named Local Authentication. It provides a default view that prompts the user to place a finger to the iPhone’s button for scanning. If that fails or users don’t want to use it, the framework allos developers to provide their own, custom made view so users can enter credentials, and provide that way an alternative for accessing the application. The framework also allows developers to set a custom string literal describing the reason of asking authentication. Besides that, it supports a variety of errors types, where each one represents a fail reason and gives the option to take proper action in each case. The error types are implemented as an enum.

### **1.1 Support biometric authentication**

Import the LocalAuthentication framework to the app.

Go to Project Settings > General and scroll down to the Linked Frameworks and Libraries section.

A screenshot of a cell phone

Description automatically generated

Click on the + sign and add LocalAuthentication.framework.

A screenshot of a cell phone

Description automatically generated

**Whenever possible, support biometric authentication.** Face ID and Touch ID are secure, familiar authentication methods that people trust.

### **1.2 How to use TouchID**

Create an LAContext object , which will provide a UI for evaluating the authentication policies and access controls, managing credentials, and invalidating authentication contexts.

Don’t forget to add the **#available(iOS 8.0, macOS 10.12.1, \*)** before adding any code for TouchID/FaceID authentication. They are not supported below iOS 8. After adding the case we have to check whether a particular policy can be evaluated, here its device authentication with bio-metrics. When that block is complete we can run the **evaluatePolicy** to get user authentication/consent. Once user authenticate go ahead and modify the UI or do your action.

After a successful authentication is time to update the UI part, remember always use **DispatchQueue.main.async** to run those tasks. The UI Changes must run in main thread.

Add the following

**import** UIKit

**import** LocalAuthentication

**class** ViewController: UIViewController {

**override** **func** viewDidLoad() {

**super**.viewDidLoad()

authenticationWithTouchID()

}

}

**extension** ViewController {

**func** authenticationWithTouchID() {

**let** localAuthenticationContext = LAContext()

localAuthenticationContext.localizedFallbackTitle = "Use Passcode"

**var** authError: NSError?

**let** reasonString = "To access the secure data"

**if** **#available**(**iOS** 8.0, **macOS** 10.12.1, \*) {

**if** localAuthenticationContext.canEvaluatePolicy(.deviceOwnerAuthenticationWithBiometrics, error: &authError) {

localAuthenticationContext.evaluatePolicy(.deviceOwnerAuthenticationWithBiometrics, localizedReason: reasonString) { success, evaluateError **in**

DispatchQueue.main.async {

**if** success {

//TODO: User authenticated successfully, take appropriate action

} **else** {

//TODO: User did not authenticate successfully, look at error and take appropriate action

**guard** **let** error = evaluateError **else** { **return** }

print(**self**.evaluateAuthenticationPolicyMessageForLA(errorCode: error.\_code))

//TODO: If you have choosen the 'Fallback authentication mechanism selected' (LAError.userFallback). Handle gracefully

}

}

print(**self**.evaluateAuthenticationPolicyMessageForLA(errorCode: error.\_code))

//TODO: If you have choosen the 'Fallback authentication mechanism selected' (LAError.userFallback). Handle gracefully

}

}

} **else** {

**guard** **let** error = authError **else** {

**return**

}

//TODO: Show appropriate alert if biometry/TouchID/FaceID is lockout or not enrolled

print(**self**.evaluateAuthenticationPolicyMessageForLA(errorCode: error.code))

}

}

}

**func** evaluatePolicyFailErrorMessageForLA(errorCode: Int) -> String {

**var** message = ""

**if** **#available**(**iOS** 11.0, **macOS** 10.13, \*) {

**switch** errorCode {

**case** LAError.biometryNotAvailable.rawValue:

message = "Authentication could not start because the device does not support biometric authentication."

**case** LAError.biometryLockout.rawValue:

message = "Authentication could not continue because the user has been locked out of biometric authentication, due to failing authentication too many times."

**case** LAError.biometryNotEnrolled.rawValue:

message = "Authentication could not start because the user has not enrolled in biometric authentication."

**default**:

message = "Did not find error code on LAError object"

}

} **else** {

**switch** errorCode {

**case** LAError.touchIDLockout.rawValue:

message = "Too many failed attempts."

**case** LAError.touchIDNotAvailable.rawValue:

message = "TouchID is not available on the device"

**case** LAError.touchIDNotEnrolled.rawValue:

message = "TouchID is not enrolled on the device"

**default**:

message = "Did not find error code on LAError object"

}

}

**return** message;

}

**func** evaluateAuthenticationPolicyMessageForLA(errorCode: Int) -> String {

**var** message = ""

**switch** errorCode {

**case** LAError.authenticationFailed.rawValue:

message = "The user failed to provide valid credentials"

**case** LAError.appCancel.rawValue:

message = "Authentication was cancelled by application"

**case** LAError.invalidContext.rawValue:

message = "The context is invalid"

**case** LAError.notInteractive.rawValue:

message = "Not interactive"

**case** LAError.passcodeNotSet.rawValue:

message = "Passcode is not set on the device"

**case** LAError.systemCancel.rawValue:

message = "Authentication was cancelled by the system"

**case** LAError.userCancel.rawValue:

message = "The user did cancel"

**case** LAError.userFallback.rawValue:

message = "The user chose to use the fallback"

**default**:

message = evaluatePolicyFailErrorMessageForLA(errorCode: errorCode)

}

**return** message

}

}

### **1.3 How to test**

Try testing in a real device or in the simulator (preferable iPhone 8).

A screenshot of a cell phone

Description automatically generated

### **1.4 Error types**

The error types are implemented as an enum:

**enum** LAError : Int {

**case** AuthenticationFailed

**case** UserCancel

**case** UserFallback

**case** SystemCancel

**case** PasscodeNotSet

**case** TouchIDNotAvailable

**case** TouchIDNotEnrolled

}

Each one of the above error types has its own significance, and it’s obvious that we *must* handle every case.

* *AuthenticationFailed*: This error type is returned when the user doesn’t provide the correct credentials, for example when using a wrong finger.
* *UserCancel*: This one is returned when the user deliberately cancels the authentication process.
* *UserFallback*: It’s returned in case the user intentionally doesn’t use the TouchID authentication and falls back to the custom way of entering credentials.
* *SystemCancel*: In this case, the system stops the authentication process because another application became active.
* *PasscodeNotSet*: When no passcode has been set by the user in the device’s Settings.
* *TouchIDNotAvailable*: This one regards the lack of TouchID because it’s not supported by the device.
* *TouchIDNotEnrolled*: It’s returned when the TouchID is supported, but there are no enrolled fingers.

### **1.5 Optional error**

Note that the error variable is declared as an optional, because if there’s no error it will be nil.

**var** authError: NSError?

Let me remind you that the nil in Swift is different from the nil in Objective-C, and it means that there’s no value at all. Also, for the *reasonString* variable I omitted its type, as the compiler will infer it from the assigned value. The *reasonString* can be customised, so feel free to set any message you like to it.

The next step is to ask the framework if the TouchID authentication can be applied to the specific device, by calling a special function named *canEvaluatePolicy*. It accepts two parameters, the policy we want to evaluate and an error object.

### **1.6 Ask for authentication**

The next step is to ask the framework if the TouchID authentication can be applied to the specific device, by calling a special function named *canEvaluatePolicy*. It accepts two parameters, the policy we want to evaluate and an error object.

// Check if the device can evaluate the policy.

**if** localAuthenticationContext.canEvaluatePolicy ( .deviceOwnerAuthenticationWithBiometrics, error: &authError) {

}

The *DeviceOwnerAuthenticationWithBiometrics* is a property of the *LAPolicy* class. Note that the error variable is passed by reference. If this condition is true, then the device supports the TouchID authentication, the TouchID mechanism has been enabled in the device’s Settings, a passcode has been also set, and of course, one finger at least has been enrolled. That means that the specific authentication policy can be applied, and the TouchID authentication dialog to appear:

// Check if the device can evaluate the policy.

**if** context.canEvaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiometrics, error: &error) {

[context .evaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiometrics, localizedReason: reasonString, reply: { (success: Bool, evalPolicyError: NSError?) -> Void **in**

// code goes here

})]

}

### **1.7 Evaluate Policy with Success or Error**

The *evaluatePolicy* accepts three arguments, where the third one is a completion handler block. In case of a successful authentication, we will load the notes from the disk (we’ll do it later). If any error occurs though, it must be handled. Actually, this is just a tutorial and not a real application, therefore we are only going to display some error messages. Note that among the possible errors there is the user’s choice to fallback to the custom authentication and avoid the fingerprint scanning, so, when it’s appropriate, we will call another method that we can implement later for showing a custom alert view to allow users to enter their password.

// Check if the device can evaluate the policy.

**if** context.canEvaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiometrics, error: &error) {

[context .evaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiometrics, localizedReason: reasonString, reply: { (success: Bool, evalPolicyError: NSError?) -> Void **in**

**if** success {

}

**else**{

// If authentication failed then show a message to the console with a short description.

// In case that the error is a user fallback, then show the password alert view.

println(evalPolicyError?.localizedDescription)

**switch** evalPolicyError!.code {

**case** LAError.SystemCancel.toRaw():

println("Authentication was cancelled by the system")

**case** LAError.UserCancel.toRaw():

println("Authentication was cancelled by the user")

**case** LAError.UserFallback.toRaw():

println("User selected to enter custom password")

NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void **in**

**self**.showPasswordAlert()

})

**default**:

println("Authentication failed")

NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void **in**

**self**.showPasswordAlert()

})

}

}

})]

}

If the *success* parameter of the completion handler is true, then we’ll load the notes data. However, if there’s an error, then we do two things: At first, we display a description of that error to the console. The *evalPolicyError* parameter value is an optional, therefore the question mark (?) symbol is necessary to be used so as the error value to be unwrapped. Then in a switch statement we examine all the possible error cases (you can use an if statement if you want as well). It’s necessary to mention here two facts: The first one is that not all error types exist here, as some of them can happen when asking the Local Authenticationframework if the TouchID authentication can be applied using the canEvaluatePolicy method, and we are about to face them right next. The second is that we use the toRaw() method along with every error type, because we want each error type to be converted from an enum value to a raw integer value. If we wouldn’t use it, the compiler would issue an error (feel free to try it).

### **1.8 Alert**

Implement the *showPasswordAlertMethod*.

As you assume, when it’s called, the custom alert for typing a password is appeared. Notice that we call it in two cases: When the user decides to fallback to the custom password entry, and when the authentication fails because the TouchID mechanism cannot recognise the user’s fingerprint. The important here is that we execute this method on the main thread, because the appearance of the alert view is meaning a visual change on the app and this can’t be done to a secondary thread, in which the completion handler is executed. Anyway, in the other two cases we just display messages to the console.

Hint:

**func** showPasswordAlert() {

**let** ac = UIAlertController(title: "Touch ID not available", message: "Your device is not configured for Touch ID.", preferredStyle: .alert)

ac.addAction(UIAlertAction(title: "OK", style: .default))

present(ac, animated: **true**)

}

### **1.9 Possibilities**

The actual act of authenticating users has a number of possible results, and you need to catch them all:

* The user might not have a Touch ID-capable device.
* The user might have a Touch ID-capable device, but might not have configured it.
* The user failed to authenticate, perhaps because they asked to enter a passcode rather than use Touch ID.

Note that Apple insists that your app provide a passcode method of authentication as a back up. More annoyingly, you need to request and store this passcode yourself – it's not even done by Apple using the system unlock code!

## Face ID

* On 12th September 2017, Apple introduced a new flagship phone the iPhoneX with a powerful new login system. Because phone’s all-glass front leaves no room for a home button, Apple is ditching TouchID in favor of a facial recognition system powered by a new camera array and a specially modified A11 chip.
* FaceID also works with the new framework called Local Authentication, wich provides facilities for requesting authentication from users with specified security policies.

### **2.1 Same here**

Try the same code (or logic) as presented with TouchID but this time using FaceID. Hint: use an iPhoneX real device.