Skip to Content

- Explore
  - [iOS & Swift Learn iOS development in Swift]

- [Android & Kotlin Learn Android development in Kotlin]()

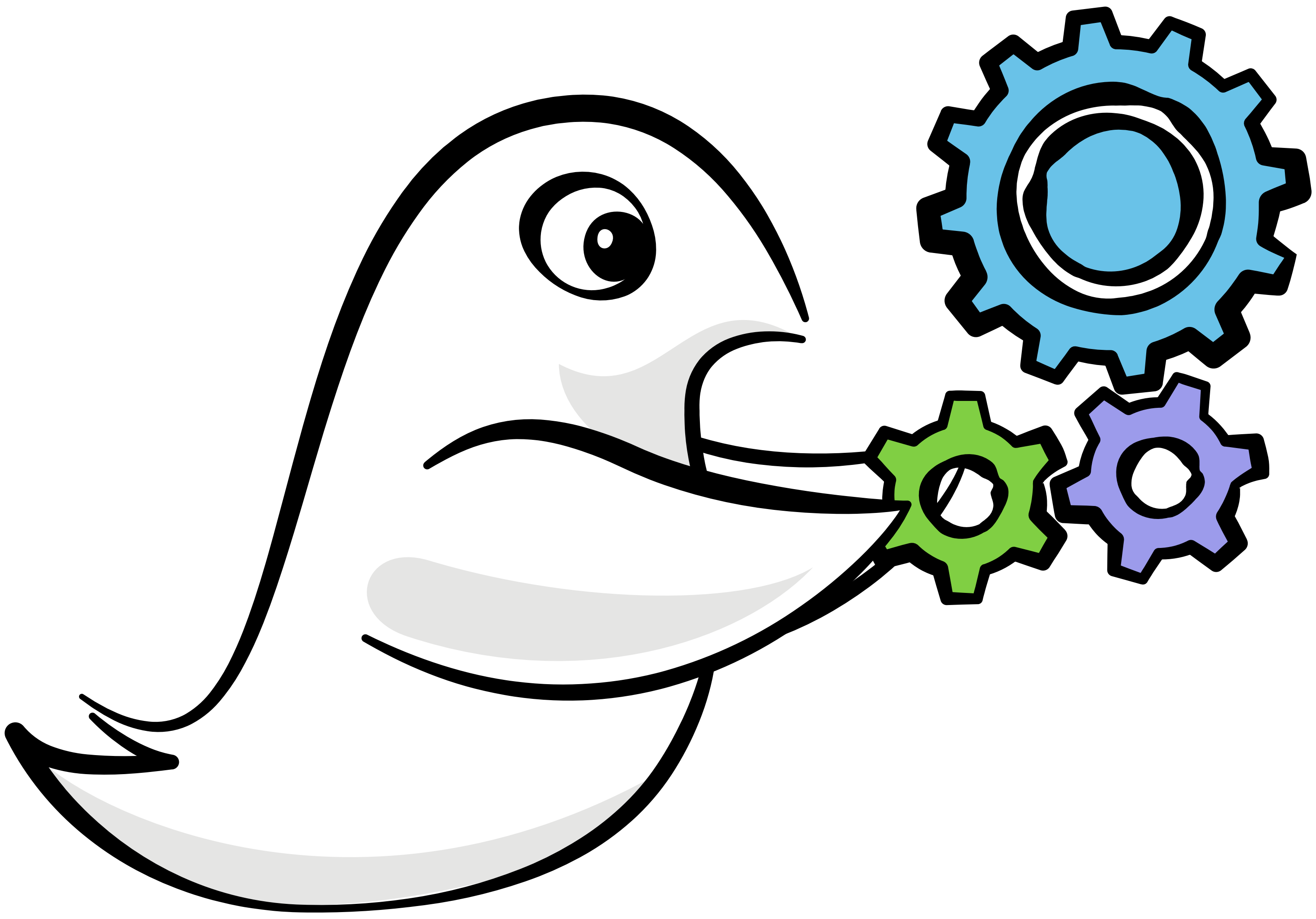
- Unity Beta

- ◦ Unreal Engine Beta
  ◦ Explore all of raywenderlich.com
- Paths
- Store
- Search
- ◦ Night Mode (Off) ▢

●●●

-
    - Newsletter
    - Forums
    - Night Mode (Off) ☐

Sign In
Create a free account



iOS & Swift Tutorials

Learn iOS development in Swift. Over 2,000 high quality tutorials!

Follow

In-App Purchase Tutorial: Getting Started

Learn how to grow app revenue in this in-app purchase tutorial by allowing users to purchase or unlock content or features.

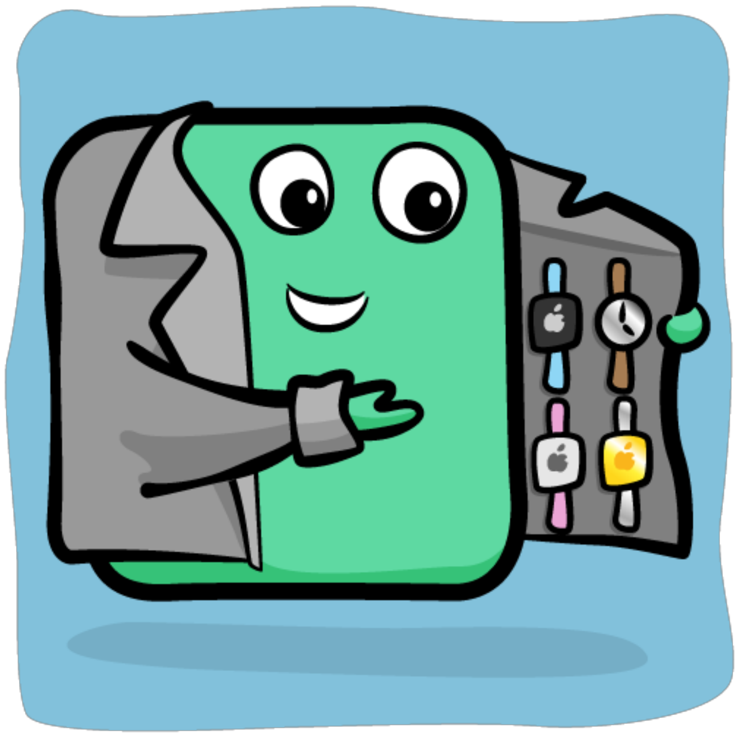
Pietro Rea Jul 25 2018 · Intermediate · Article · 30 mins

Download Materials

Bookmark

Version

- Swift 4.2, iOS 12, Xcode 10



Update note: Pietro Rea updated this tutorial for Xcode 10, Swift 4.2 and iOS 11/12. Ray Wenderlich wrote the original.

One of the great things about building iOS apps is that you have lots of choices when it comes to monetizing your app: plain vanilla paid apps, free apps supported by ads, or even apps that support in-app purchases.

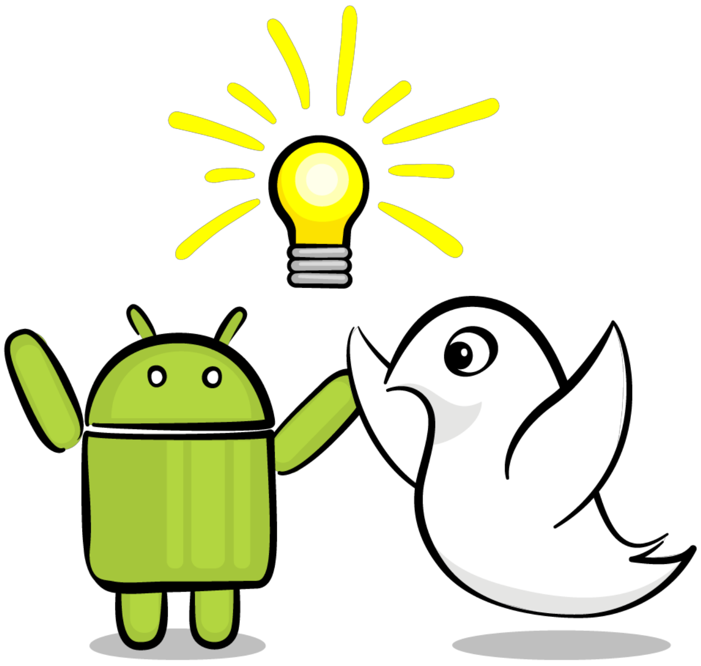
An *in-app purchase* (or *IAP*) allows developers to charge users for specific functionality or content while using an app. Implementing IAPs is particularly compelling for several reasons:

- It's an extra way to earn money, in addition to simply selling the app for a fee upfront. Some users are willing to spend a lot more on extra content or features.
- An app can be offered for free, which makes it a no-brainer download for most people. Free apps will typically get *many* more downloads than paid apps. If users enjoy the app, then they can purchase more content or functionality later.
- You can display advertisements to the user in a free app with an option to remove them by purchasing an IAP.
- Following the initial release of an app, new paid content can be added to the same app instead of having to develop a brand new app to earn more money.

In this in-app purchase tutorial, you'll leverage IAPs to unlock extra content embedded in an app. You'll need to be familiar with basic Swift and iOS programming concepts. If these are unfamiliar topics, then check out our range of Swift tutorials before getting started. You'll also need a paid developer account, with access to both the iOS Developer Center and App Store Connect.

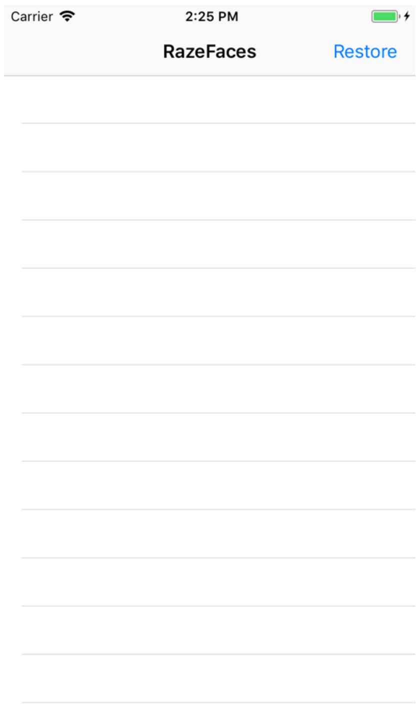
Getting Started

In this in-app purchase tutorial, you'll build a small app called "RazeFaces", which allows users to buy a "RazeFace", which is a neat illustration commonly used on this site for books and videos.



A typical "RazeFace"

Download the materials using the link at the top and open the starter project in Xcode. Build and run to see what it does so far. The answer is: Not a lot! You'll see an empty table view, with a single *Restore* button in the navigation bar, which will be hooked up later to restore purchases.



Upon finishing this tutorial, there will be a list of RazeFaces listed in the table view which you'll be able to buy. If you delete and reinstall the app, the *Restore* button will reinstate any previously purchased RazeFaces.

Head over to Xcode to take a quick look at the code. The main view controller is in *MasterViewController.swift*. This class displays the table view which will contain a list of available IAPs. Purchases are stored as an array of `SKProduct` objects.

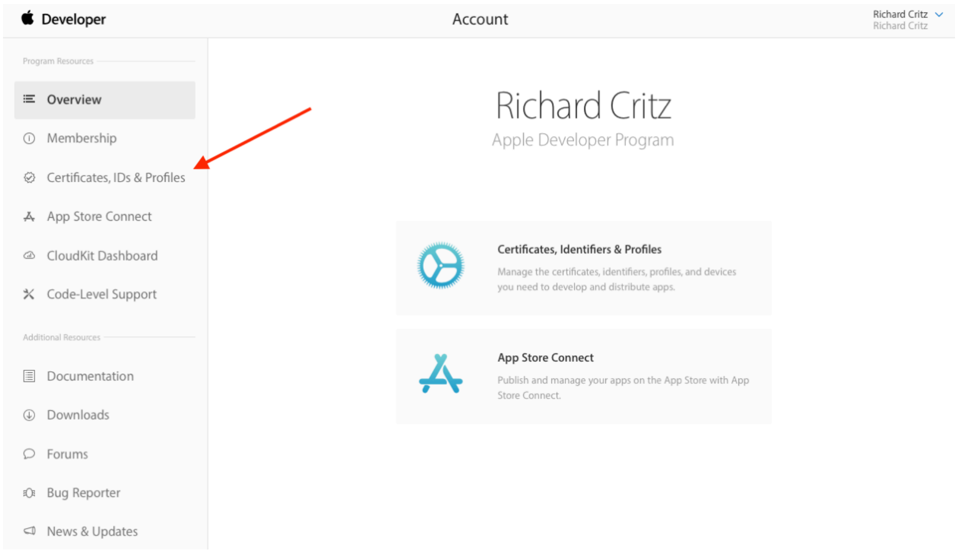
Notice that `MasterViewController` is using an object called `RazeFaceProducts.store` of type `IAPHelper` to do the heavy lifting. Take a look at their respective code files, *RazeFaceProducts.swift* and *IAPHelper.swift*.

`RazeFaceProducts` is a simple struct that contains some information about the products in the app, and `IAPHelper` does all the important work of talking to *StoreKit*. The methods are all stubbed out at the moment, but you'll fill them out in this tutorial to add IAP functionality to the app.

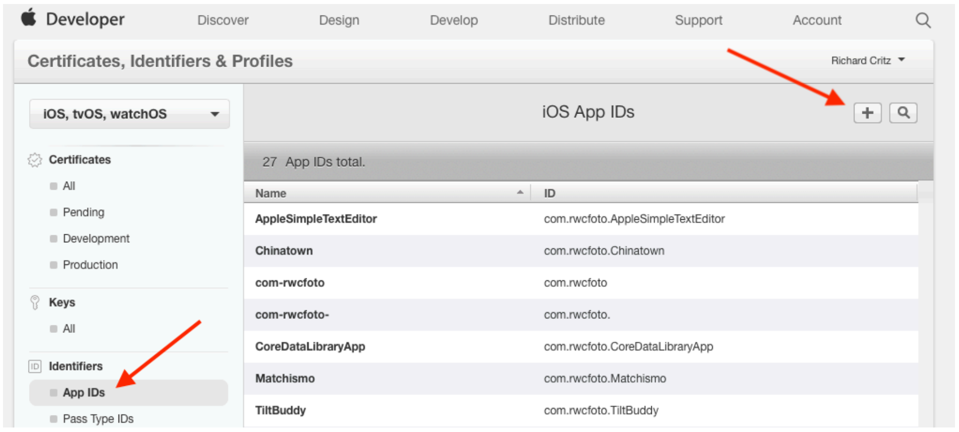
Before writing any code to incorporate IAP, you'll first need to do some setup in the iOS Developer Center and App Store Connect.

Creating an App ID

First, you need to create an App ID. This will link together your app to your in-app purchaseable products. Login to the Apple Developer Center, then select *Certificates, IDs & Profiles*.



Next, select *Identifiers* > *App IDs*, and click + in the upper right corner to create a new App ID.



27 App IDs total.

Name	ID
AppleSimpleTextEditor	com.rwcfoto.AppleSimpleTextEditor
Chinatown	com.rwcfoto.Chinatown
com-rwcfoto	com.rwcfoto
com-rwcfoto-	com.rwcfoto.
CoreDataLibraryApp	com.rwcfoto.CoreDataLibraryApp
Matchismo	com.rwcfoto.Matchismo
TiltBuddy	com.rwcfoto.TiltBuddy

Fill out the information for the new App ID. Enter *RazeFace IAP Tutorial App* for the *Name*. Choose *Explicit App ID* and enter a unique *Bundle ID*. A common practice is to use your domain name in reverse (for example, *com.razeware.razefaces*). Make note of the Bundle ID as it will be needed in the steps that follow.

Scroll down to the *App Services* section. Notice that *In-App Purchase* and *GameCenter* are enabled by default. Click *Continue* and then *Register* and *Done*.

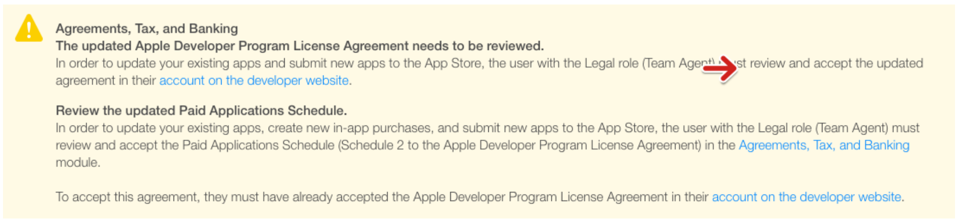
Congratulations! You have a new App ID! Next, you'll create a matching app in App Store Connect.

Checking Your Agreements

Before you can add IAPs to an app in iTunes Connect, you must do two things:

- Make sure you have accepted the latest Apple Development Program License Agreement on developer.apple.com.
- Make sure you have accepted the latest Paid Applications agreement in the Agreements, Tax, and Billing section in App Store Connect.

If you have not done this, usually iTunes Connect will give you a warning like the following:



Agreements, Tax, and Banking
The updated Apple Developer Program License Agreement needs to be reviewed.
In order to update your existing apps and submit new apps to the App Store, the user with the Legal role (Team Agent) must review and accept the updated agreement in their account on the developer website.

Review the updated Paid Applications Schedule.
In order to update your existing apps, create new in-app purchases, and submit new apps to the App Store, the user with the Legal role (Team Agent) must review and accept the Paid Applications Schedule (Schedule 2 to the Apple Developer Program License Agreement) in the Agreements, Tax, and Banking module.

To accept this agreement, they must have already accepted the Apple Developer Program License Agreement in their account on the developer website.

If you see something like the above, follow the steps to accept the appropriate agreements.

It's also good to double check the *Agreements, Tax, and Banking* section in iTunes Connect:

Agreements, Tax, and Banking

If you see a section entitled *Request Contracts* containing a row for *Paid Applications*, then click the *Request* button. Fill out all the necessary information and submit it. It may take some time for your request to be approved. Sit tight!

Otherwise, if you see *Paid Applications* listed under *Contracts In Effect*, then it looks like you've already done this step! Nice job!

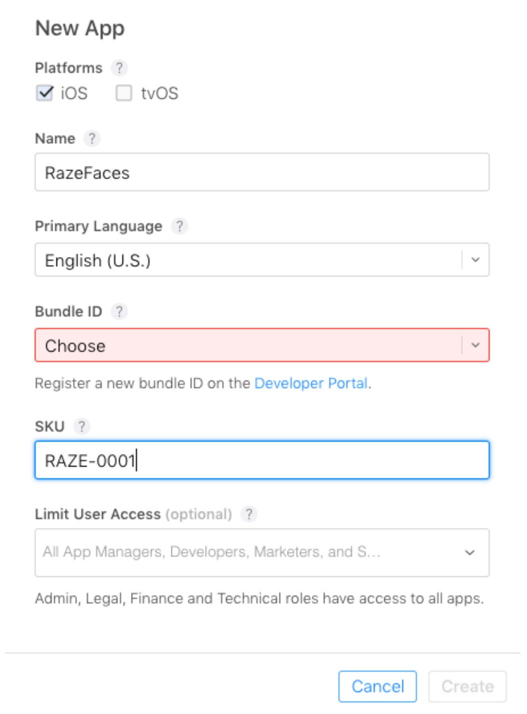
Note: Apple can take days to approve these IAP-related agreements after you submit them. During this time, you won't be able to display IAP products in your apps even if you implement everything correctly in code. This is a common source of frustration for folks implementing In-App Purchases for the first time. Hang in there!

Creating an App in iTunes Connect

Now to create the app record itself, click *App Store Connect* in the upper left corner of the page, then click *My Apps*.

My Apps

Next, click + in the upper left corner of the page and select *New App* to add a new app record. Fill out the information as shown here:


New App

Platforms

[✓ iOS] tvOS

Name: RazeFaces

Primary Language: English (U.S.)

Bundle ID: Choose

Register a new bundle ID on the Developer Portal.

SKU: RAZE-0001

Limit User Access (optional): All App Managers, Developers, Marketers, and S...

Admin, Legal, Finance and Technical roles have access to all apps.

You won't be able to use the exact same app *Name* that you see here, because app names need to be unique across the App Store. Perhaps add your own initials after the example title shown in the screenshot above.

Note: If you are quick in getting to this step, the Bundle ID might not be showing up in the dropdown list. This sometimes takes a while to propagate through Apple's systems.

Click *Create* and you're done!

Creating In-App Purchase Products

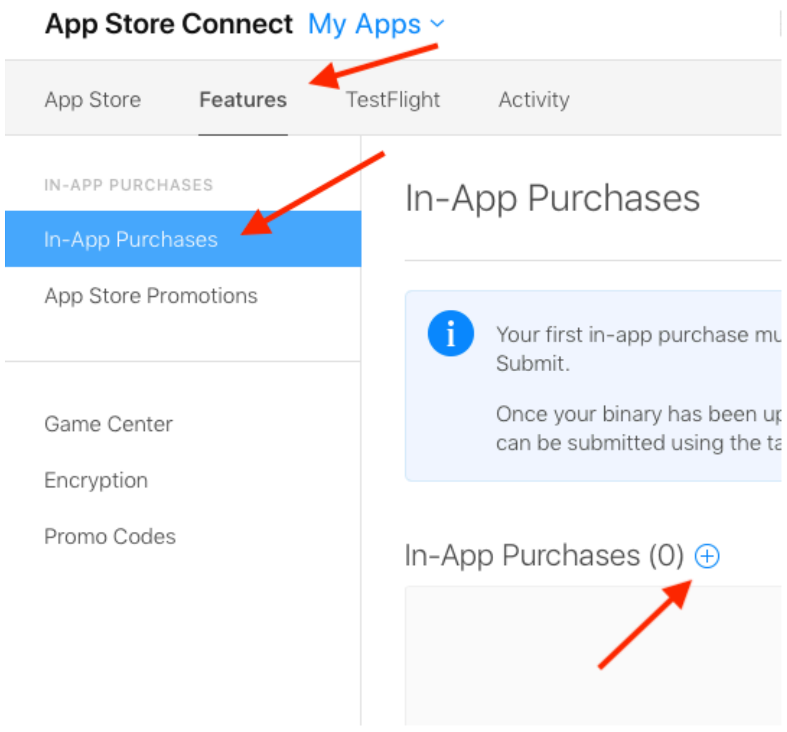
When offering IAPs you must first add an entry for each individual purchase within App Store Connect. If you've ever listed an app for sale in the store, it's a similar process and includes things like choosing a pricing tier for the purchase. When the user makes a purchase, the App Store handles the complex process of charging the user and reply with data about such operation.

There are a whole bunch of different types of IAP you can add:

- *Consumable*: These can be bought more than once and can be used up. These are a good fit for extra lives, in-game currency, temporary power-ups, and the like.
- *Non-Consumable*: Something that you buy once, and expect to have permanently such as extra levels and unlockable content. The RazeFace illustrations from this tutorial fall into this category.
- *Non-Renewing Subscription*: Content that's available for a fixed period of time.
- *Auto-Renewing Subscription*: A repeating subscription such as a monthly raywenderlich.com subscription.

You can only offer In-App Purchases for digital items, and not for physical goods or services. For more information about all of this, check out Apple's full documentation on Creating In-App Purchase Products.

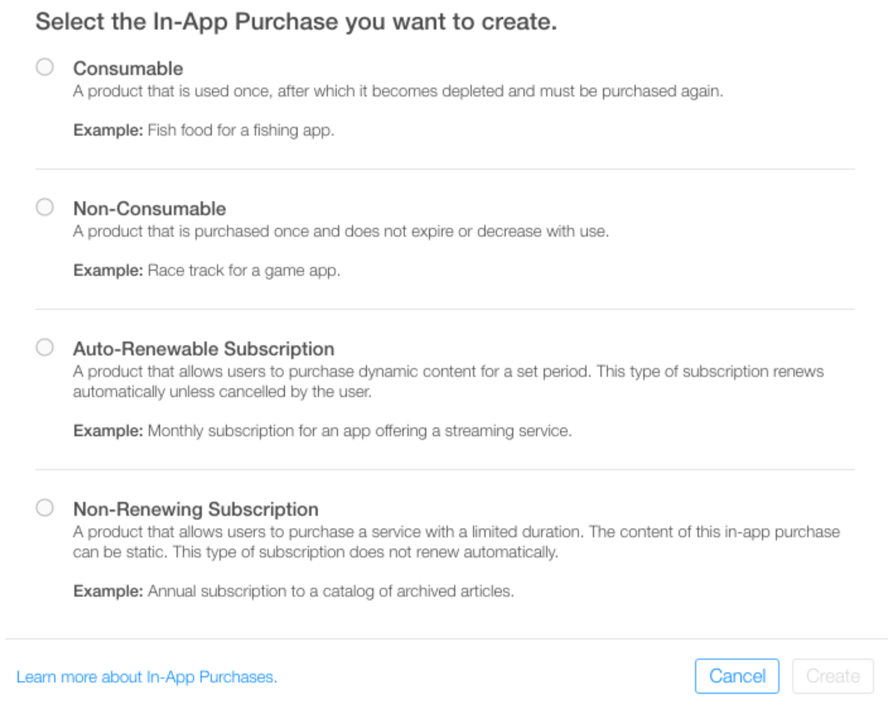
Now, while viewing your app's entry in App Store Connect, click on the *Features* tab and then select *In-App Purchases*. To add a new IAP product, click the + to the right of *In-App Purchases*.



Your first in-app purchase mu
Submit.

Once your binary has been up
can be submitted using the ta

You'll see the following dialog appear:



Select the In-App Purchase you want to create.

Consumable
A product that is used once, after which it becomes depleted and must be purchased again.

Example: Fish food for a fishing app.

Non-Consumable
A product that is purchased once and does not expire or decrease with use.

Example: Race track for a game app.

Auto-Renewable Subscription
A product that allows users to purchase dynamic content for a set period. This type of subscription renews automatically unless cancelled by the user.

Example: Monthly subscription for an app offering a streaming service.

Non-Renewing Subscription
A product that allows users to purchase a service with a limited duration. The content of this in-app purchase can be static. This type of subscription does not renew automatically.

Example: Annual subscription to a catalog of archived articles.

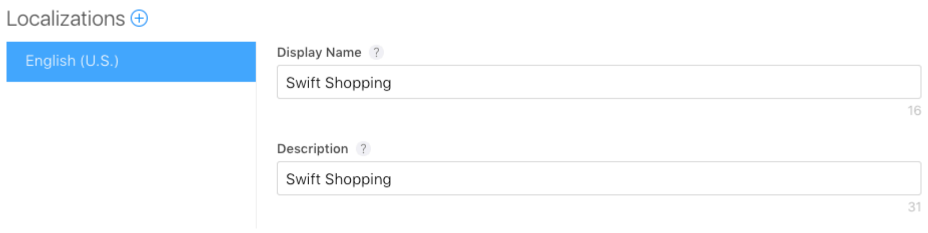
When a user purchases a RazeFace in your app, you'll want them to always have access to it, so select *Non-Consumable*, and click *Create*.

Next, fill out the details for the IAP as follows:

- *Reference Name*: A nickname identifying the IAP within iTunes Connect. This name does not appear anywhere in the app. The title of the RazeFace you'll be unlocking with this purchase is *Swift Shopping*, so enter that here.
- *Product ID*: This is a unique string identifying the IAP. Usually it's best to start with the Bundle ID and then append a unique name specific to this purchasable item. For this tutorial, make sure you append *swiftshopping*, as this will be used later within the app to look up the RazeFace to unlock. For example, you can use: *com.theNameYouPickedEarlier.razefaces.swiftshopping*.
- *Cleared for Sale*: Enables or disables the sale of the IAP. You want to enable it!

- *Price Tier*: The cost of the IAP. Choose *Tier 1*.

Now scroll down to the *Localizations* section and note that there is a default entry for English (U.S.). Enter “Swift Shopping” for both the *Display Name* and the *Description*. Click *Save*. Great! You’ve created your first IAP product.



Display Name: Swift Shopping
Description: Swift Shopping

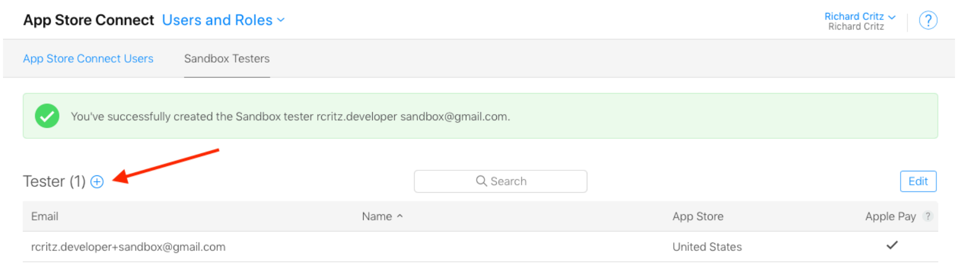
Note: App Store Connect may complain that you’re missing metadata for your IAP. Before you submit your app for review, you’re required to add a screenshot of the IAP at the bottom of this page. The screenshot is used *only* for Apple’s review and does not appear in your App Store listing.

There’s one more step required before you can delve into some code. When testing in-app purchases in a development build of an app, Apple provides a test environment which allows you to “purchase” your IAP products without creating financial transactions.

These special test purchases can only be made by a special “Sandbox Tester” user account in App Store Connect. You’re almost at the code, I promise!

Creating a Sandbox User

In App Store Connect, click *App Store Connect* in the top left corner of the window to get back to the main menu. Select *Users and Roles*, then click the *Sandbox Testers* tab. Click + next to the “Tester” title.



You've successfully created the Sandbox tester rcritz.developer sandbox@gmail.com.

Email	Name	App Store	Apple Pay
rcritz.developer+sandbox@gmail.com		United States	✓

Fill out the information and click *Save* when you’re done. You can make up a first and last name for your test user, but you must use a real email address as Apple will send a verification email to the address. Once you receive that email, be sure to click the link in it to verify your address.

The email address you enter should also NOT already be associated with an Apple ID account. Hint: if you have a gmail account, you can simply use an address alias instead of having to create a brand new account.

Note: Unfortunately, testing a new purchase of a *non-consumable* IAP requires a new sandbox tester (and email address) each time. Repeated purchases using the same sandbox tester will be treated as restoring an already purchased item, so any code specific to new purchases will not be exercised.

If multiple test runs through new purchase code are necessary and your email provider does not support qualifiers, then consider setting up a consumable IAP just for testing purposes. Delete the app on your device after each test and the purchase of a consumable IAP will be considered a new purchase.

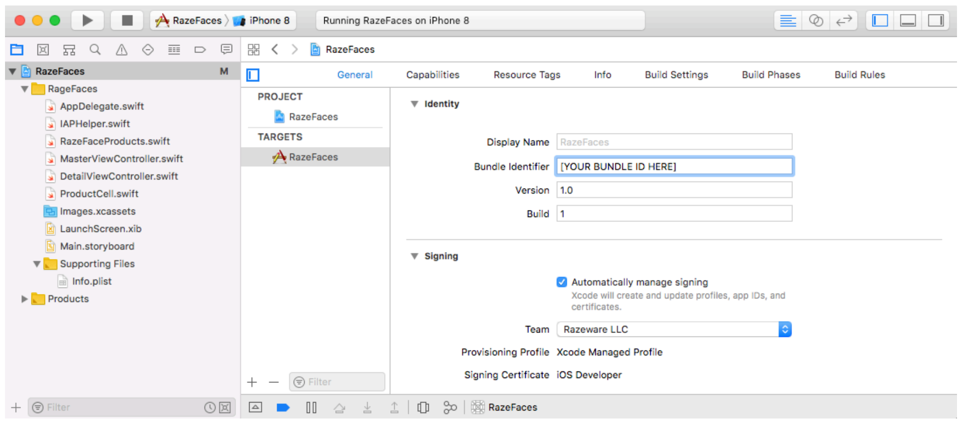
One strategy you could adopt is testing the failure cases as many times as possible before testing the successful case. That way you’ll need to create fewer sandbox testers. In general, remember the rule that once a user (even a sandbox one) has bought a *non-consumable* IAP, he can’t buy it again, only restore it.

Great — you now have a test user. You can finally implement IAPs in your app!

Project Configuration

For everything to work correctly, it’s really important that the bundle identifier and product identifiers in the app match the ones you just created in the Developer Center and in App Store Connect.

Head over to the starter project in Xcode. Select the *RazeFaces* project in the Project navigator, then select it again under *Targets*. Select the *General* tab, switch your *Team* to your correct team, and enter the bundle ID you used earlier.



Identity

Display Name: RazeFaces
Bundle Identifier: [YOUR BUNDLE ID HERE]
Version: 1.0
Build: 1

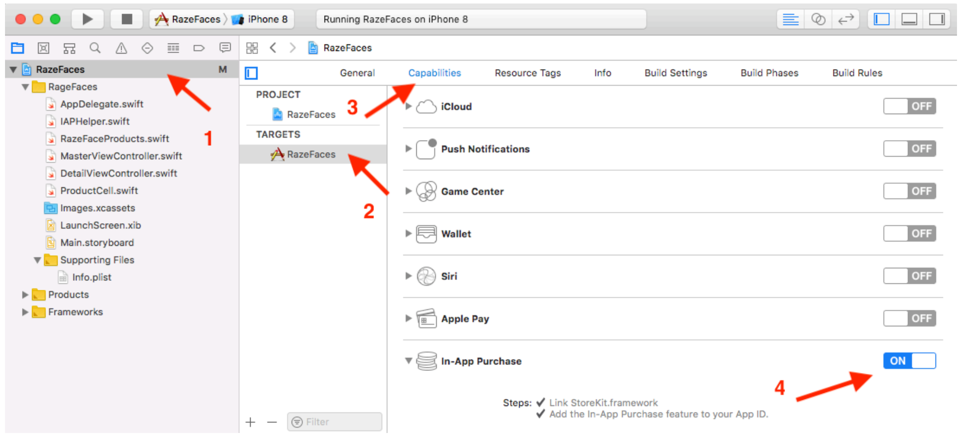
Signing

✓ Automatically manage signing
Xcode will create and update profiles, app IDs, and certificates.

Team: Razeware LLC
Provisioning Profile: Xcode Managed Profile
Signing Certificate: iOS Developer

Next select the *Capabilities* tab. Scroll down to *In-App Purchase* and toggle the switch to *ON*.

Note: If IAP does not show up in the list, make sure that, in the Accounts section of Xcode preferences, you are logged in with the Apple ID you used to create the app ID.



iCloud: OFF
Push Notifications: OFF
Game Center: OFF
Wallet: OFF
Siri: OFF
Apple Pay: OFF
In-App Purchase: ON

Steps: ✓ Link StoreKit.framework
✓ Add the In-App Purchase feature to your App ID.

Open *RazeFaceProducts.swift*. Notice that there is a placeholder reference to the IAP product you created: `SwiftShopping`. Replace this with the full Product ID that you configured in App Store Connect — for example:

```
public static let SwiftShopping = "com.theNameYouPickedEarlier.razefaces.swiftshopping"
```

Note: The list of product identifiers can be pulled from a web server so new IAPs can be added dynamically rather than requiring an app update. This tutorial keeps things simple and uses hard-coded product identifiers.

Listing In-App Purchases

The `store` property of `RazeFaceProducts` is an instance of `IAPHelper`. As mentioned earlier, this object interacts with the *StoreKit* API to list and perform purchases. Your first task is to update `IAPHelper` to retrieve a list of IAPs — there's only one so far — from Apple's servers.

Open *IAPHelper.swift*. At the top of the class, add the following private property:

```
private let productIdentifiers: Set<ProductIdentifier>
```

Next, add the following to `init(productIds:)` before the call to `super.init()`:

```
productIdentifiers = productIds
```

An `IAPHelper` instance is created by passing in a set of product identifiers. This is how `RazeFaceProducts` creates its `store` instance.

Next, add these other private properties just under the one you added a moment ago:

```
private var purchasedProductIdentifiers: Set<ProductIdentifier> = []
private var productsRequest: SKProductsRequest?
private var productsRequestCompletionHandler: ProductsRequestCompletionHandler?
```

`purchasedProductIdentifiers` tracks which items have been purchased. The other two properties are used by the `SKProductsRequest` delegate to perform requests to Apple servers.

Next, still in *IAPHelper.swift* replace the implementation of `requestProducts(_:)` with the following:

```
public func requestProducts(completionHandler: @escaping ProductsRequestCompletionHandler) {
  productsRequest?.cancel()
  productsRequestCompletionHandler = completionHandler

  productsRequest = SKProductsRequest(productIdentifiers: productIdentifiers)
  productsRequest!.delegate = self
  productsRequest!.start()
}
```

This code saves the user's completion handler for future execution. It then creates and initiates a request to Apple via an `SKProductsRequest` object. There's one problem: the code declares `IAPHelper` as the request's delegate, but it doesn't yet conform to the `SKProductsRequestDelegate` protocol.

To fix this, add the following extension to the very end of *IAPHelper.swift*, after the last curly brace:

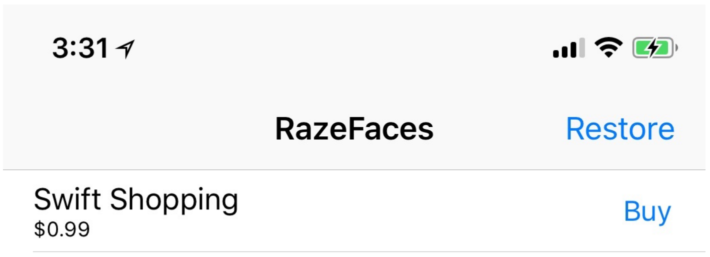
```
// MARK: - SKProductsRequestDelegate

extension IAPHelper: SKProductsRequestDelegate {

  public func productsRequest(_ request: SKProductsRequest, didReceive response: SKProductsResponse) {
    print("Loaded list of products...")
    let products = response.products
    productsRequestCompletionHandler?(true, products)
    clearRequestAndHandler()

    for p in products {
      print("Found product: \(p.productIdentifier) \(p.localizedTitle) \(p.price.floatValue)")
    }
  }

  public func request(_ request: SKRequest, didFailWithError error: Error) {
    print("Failed to load list of products.")
    print("Error: \(error.localizedDescription)")
    productsRequestCompletionHandler?(false, nil)
    clearRequestAndHandler()
  }

  private func clearRequestAndHandler() {
    productsRequest = nil
    productsRequestCompletionHandler = nil
  }
}
```

This extension is used to get a list of products, their titles, descriptions and prices from Apple's servers by implementing the two methods required by the `SKProductsRequestDelegate` protocol.

`productsRequest(_:didReceive:)` is called when the list is succesfully retrieved. It receives an array of `SKProduct` objects and passes them to the previously saved completion handler. The handler reloads the table with new data. If a problem occurs, `request(_:didFailWithError:)` is called. In either case, when the request finishes, both the request and completion handler are cleared with `clearRequestAndHandler()`.

Build and run. Hooray! A list of products (only one so far) is displayed in the table view! It took some work, but you got there in the end.

Note: You can display IAP products on both the iOS simulator as well as physical iOS devices, but if you want to test buying or restoring purchases, you can only do this on physical devices. More on this in the purchasing section below.



Note: If the run was unsuccessful and you didn't see any products, then there are a number of things to check. This list is courtesy of *itsme.manish* and *abgtan* from the forums on earlier versions of this post, plus more tips added over time.

- Does the project's Bundle ID match the App ID from the iOS Development Center?
- Is the full product ID being used when making an `SKProductRequest`? (Check the `productIdentifiers` property of `RazeFaceProducts`.)
- Is the Paid Applications Contract in effect on iTunes Connect? It can take hours to days for them to go from pending to accepted from them moment you submit them.
- Have you waited several hours since adding your product to App Store Connect? Product additions may be active immediately or may take some time.
- Check Apple Developer System Status. Alternatively, try this link. If it doesn't respond with a status value, then the iTunes sandbox may be down. The status codes are explained in Apple's Validating Receipts With the App Store documentation.
- Have IAPs been enabled for the App ID? (Did you select *Cleared for Sale* earlier?)
- Have you tried deleting the app from your device and reinstalling it?

Still stuck? As you can see, there's a lot of setting up to do for IAP. Try this tutorial's comments for a discussion with other readers.

Purchased Items

You want to be able to determine which items are already purchased. To do this, you'll use the `purchasedProductIdentifiers` property added earlier. If a product identifier is contained in this set, the user has purchased the item. The method for checking this is straightforward.

In *IAPHelper.swift*, replace the `return` statement in `isProductPurchased(_:)` with the following:

```
return purchasedProductIdentifiers.contains(productIdentifier)
```

Saving purchase status locally alleviates the need to request such data to Apple's servers every time the app starts. `purchasedProductIdentifiers` are saved using `UserDefaults`.

Still in *IAPHelper.swift*, replace `init(productIds:)` with the following:

```
public init(productIds: Set<ProductIdentifier>) {
  productIdentifiers = productIds
  for productIdentifier in productIds {
    let purchased = UserDefaults.standard.bool(forKey: productIdentifier)
    if purchased {
      purchasedProductIdentifiers.insert(productIdentifier)
      print("Previously purchased: \(productIdentifier)")
    } else {
      print("Not purchased: \(productIdentifier)")
    }
  }
  super.init()
}
```

For each product identifier, you check whether the value is stored in `UserDefaults`. If it is, then the identifier is inserted into the `purchasedProductIdentifiers` set. Later, you'll add an identifier to the set following a purchase.

Note: User defaults may not be the best place to store information about purchased products in a real application. An owner of a jailbroken device could easily access your app's `UserDefaults` plist, and modify it to 'unlock' purchases. If this sort of thing concerns you, then it's worth checking out Apple's documentation on Validating App Store Receipts — this allows you to verify that a user has made a particular purchase.

Making Purchases (Show Me The Money!)

Knowing what a user has purchased is great, but you still need to be able to make the purchases in the first place! Implementing purchase capability is the logical next step.

Still in *IAPHelper.swift*, replace `buyProduct(_:)` with the following:

```
public func buyProduct(_ product: SKProduct) {
  print("Buying \(product.productIdentifier)...")
  let payment = SKPayment(product: product)
  SKPaymentQueue.default().add(payment)
}
```

This creates a payment object using an `SKProduct` (retrieved from the Apple server) to add to a payment queue. The code utilizes a singleton `SKPaymentQueue` object called `default()`. Boom! Money in the bank. Or is it? How do you know if the payment went through?

Payment verification is achieved by having the `IAPHelper` observe transactions happening on the `SKPaymentQueue`. Before setting up `IAPHelper` as an `SKPaymentQueue` transactions observer, the class must conform to the `SKPaymentTransactionObserver` protocol.

Go to the very bottom (after the last curly brace) of *IAPHelper.swift* and add the following extension:

```
// MARK: - SKPaymentTransactionObserver

extension IAPHelper: SKPaymentTransactionObserver {

  public func paymentQueue(_ queue: SKPaymentQueue,
                           updatedTransactions transactions: [SKPaymentTransaction]) {
    for transaction in transactions {
      switch transaction.transactionState {
      case .purchased:
        complete(transaction: transaction)
        break
      case .failed:
        fail(transaction: transaction)
        break
      case .restored:
        restore(transaction: transaction)
        break
      case .deferred:
        break
      case .purchasing:
        break
      }
    }
  }

  private func complete(transaction: SKPaymentTransaction) {
    print("complete...")
    deliverPurchaseNotificationFor(identifier: transaction.payment.productIdentifier)
    SKPaymentQueue.default().finishTransaction(transaction)
  }

  private func restore(transaction: SKPaymentTransaction) {
    guard let productIdentifier = transaction.original?.payment.productIdentifier else { return }

    print("restore... \(productIdentifier)")
    deliverPurchaseNotificationFor(identifier: productIdentifier)
    SKPaymentQueue.default().finishTransaction(transaction)
  }

  private func fail(transaction: SKPaymentTransaction) {
    print("fail...")
    if let transactionError = transaction.error as NSError?,
      let localizedDescription = transaction.error?.localizedDescription,
        transactionError.code != SKError.paymentCancelled.rawValue {
        print("Transaction Error: \(localizedDescription)")
      }

    SKPaymentQueue.default().finishTransaction(transaction)
  }

  private func deliverPurchaseNotificationFor(identifier: String?) {
    guard let identifier = identifier else { return }

    purchasedProductIdentifiers.insert(identifier)
    UserDefaults.standard.set(true, forKey: identifier)
    NotificationCenter.default.post(name: .IAPHelperPurchaseNotification, object: identifier)
  }
}
```

That's a lot of code! A detailed review is in order. Fortunately, each method is quite short.

`paymentQueue(_:updatedTransactions:)` is the only method actually required by the protocol. It gets called when one or more transaction states change. This method evaluates the state of each transaction in an array of updated transactions and calls the relevant helper method: `complete(transaction:)`, `restore(transaction:)` or `fail(transaction:)`.

If the transaction was completed or restored, it adds to the set of purchases and saves the identifier in `UserDefaults`. It also posts a notification with that transaction so that any interested object in the app can listen for it to do things like update the user interface. Finally, in both the case of *success* or *failure*, it marks the transaction as finished.

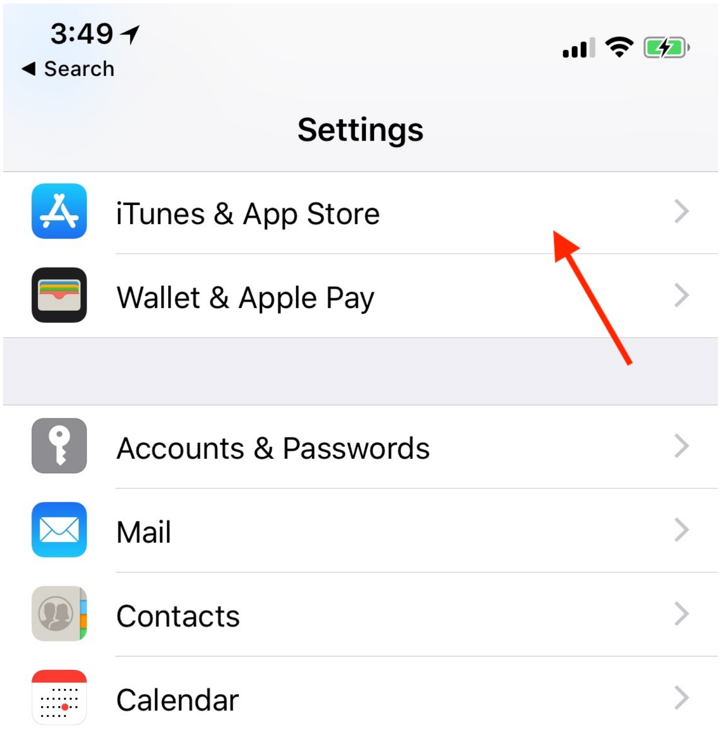
All that's left is to hook up `IAPHelper` as a payment transaction observer. Still in *IAPHelper.swift*, go back to `init(productIds:)` and add the following line right *after* `super.init()`.

```
SKPaymentQueue.default().add(self)
```

Making a Sandbox Purchase

Build and run the app — but to test out purchases, you'll have to run it on a device. The sandbox tester created earlier can be used to perform the purchase without getting charged. If only I could have a sandbox tester to do my grocery shopping :] Here's how to use the tester account:

Go to your iPhone and make sure you're logged out of your normal App Store account. To do this, go to the *Settings* app and tap *iTunes & App Store*.

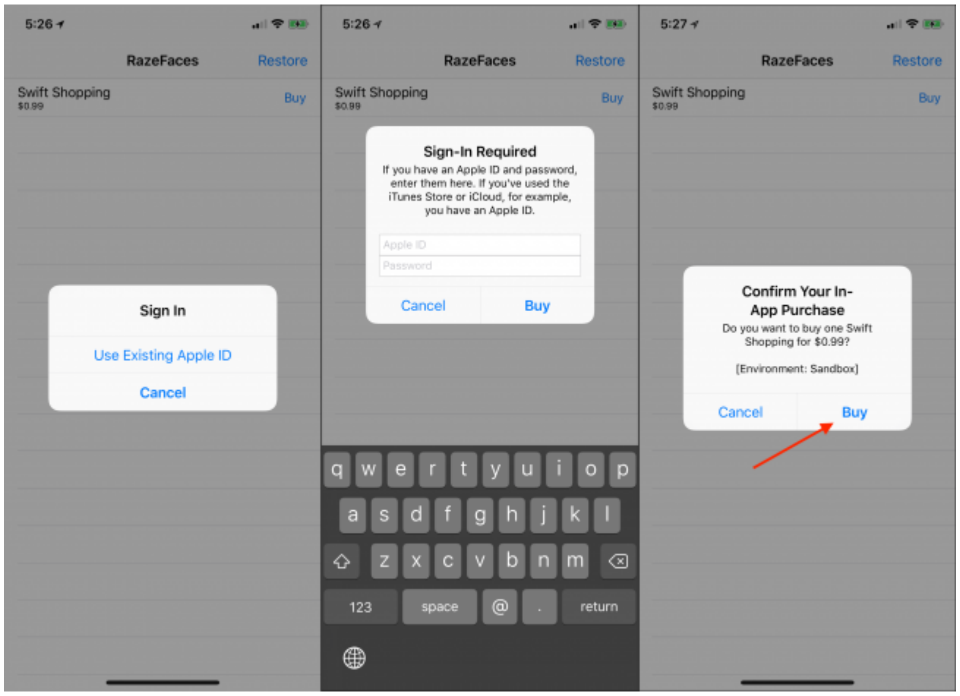


Tap your iCloud account name and then tap *Sign Out*. At this point don't actually sign in with the sandbox user. You will be prompted to do this once you attempt to buy the IAP back in the sample app.

Connect your device, build and run! You'll see your product listed in the app. To begin the purchase process, tap the *Buy* button.

An alert will appear prompting you to log in. Tap *Use Existing Apple ID*, and enter the login details for the sandbox tester account that you created earlier.

Confirm the purchase by tapping *Buy*. The alert view shows that the purchase is being made in the sandbox as a reminder that you won't be charged for it.


RazeFaces
Restore
Swift Shopping
$0.99
Buy
Sign In
Use Existing Apple ID
Cancel
Sign-In Required
If you have an Apple ID and password, enter them here. If you've used the iTunes Store or iCloud, for example, you have an Apple ID.
Apple ID
Password
Cancel
Buy
Confirm Your In-App Purchase
Do you want to buy one Swift Shopping for $0.99?
[Environment: Sandbox]
Cancel
Buy

Finally, an alert view will appear confirming the purchase was successful. Once the purchase process has been completed, a checkmark appears next to the purchased item. Tap on the *purchased item* to enjoy your new RazeFace.

Finally you get to see this "Swift Shopping" RazeFace that you've been hearing so much about!

Restoring Purchases

If the user deletes and re-installs the app or installs it on another device, then they need the ability to access previously purchased items. In fact, Apple may reject an app if it cannot restore non-consumable purchases.

As a purchase transaction observer, `IAPHelper` is already being notified when purchases have been restored. The next step is to react to this notification by restoring the purchases.

Open *IAPHelper.swift* and scroll to the bottom of the file. In the *StoreKit API extension*, replace `restorePurchases()` with the following:

```
public func restorePurchases() {
  SKPaymentQueue.default().restoreCompletedTransactions()
}
```

That was almost too easy! You've already set the transaction observer and implemented the method to handle restoring transactions in the previous step.

To test this out, after you've made a purchase in the previous step, delete the app from your device. Build and run again, then tap *Restore* on the top right. You should see a checkmark appear next to the previously purchased product.

Payment Permissions

Some devices and accounts may not permit an in-app purchase. This can happen, for example, if parental controls are set to disallow it. Apple requires this situation to be handled gracefully. Not doing so will likely result in an app rejection.

Open *IAPHelper.swift* again. In the *StoreKit API extension*, replace the `return` statement in `canMakePayments()` with this line:

```
return SKPaymentQueue.canMakePayments()
```

Product cells should behave differently depending on the value returned by `canMakePayments()`. For example, if `canMakePayments()` returns `false`, then the *Buy* button should not be shown and the price should be replaced by "Not Available".

To accomplish this, open *ProductCell.swift* and replace the entire implementation of the `product` property's `didSet` handler with the following:

```
didSet {
  guard let product = product else { return }

  textLabel?.text = product.localizedTitle

  if RazeFaceProducts.store.isProductPurchased(product.productIdentifier) {
    accessoryType = .checkmark
    accessoryView = nil
    detailTextLabel?.text = ""
  } else if IAPHelper.canMakePayments() {
    ProductCell.priceFormatter.locale = product.priceLocale
    detailTextLabel?.text = ProductCell.priceFormatter.string(from: product.price)

    accessoryType = .none
    accessoryView = self.newBuyButton()
  } else {
    detailTextLabel?.text = "Not available"
  }
}
```

This implementation will display more appropriate information when payments cannot be made with the device. And there you have it — an app with in-app purchase!

Where To Go From Here?

You can download the completed version of the project using the *Download Materials* button at the top or bottom of this tutorial. Feel free to re-use the IAP helper class in your own projects!

The In-App Purchase Video Tutorial Series by Sam Davies covers all of the topics introduced here, but goes to the next level in Part 3 where he talks about validating receipts.

One shortcoming of the sample app is that it doesn't indicate to the user when it is communicating with Apple. A possible improvement would be to display a spinner or HUD control at appropriate times. This UI enhancement, however, is beyond the scope of this tutorial. For more information on HUD controls, check out Section 3 of The iOS Apprentice.

Apple has a great landing page for in-app purchase: In-App Purchase for Developers. It collects together links to all the relevant documentation and WWDC videos.

IAPs can be an important part of your business model. Use them wisely and be sure to follow the guidelines about restoring purchases and failing gracefully, and you'll be well on your way to success!

If you have any questions or comments about this in-app purchase tutorial, then please join the forum discussion below!

Core Concepts iOS & Swift Tutorials

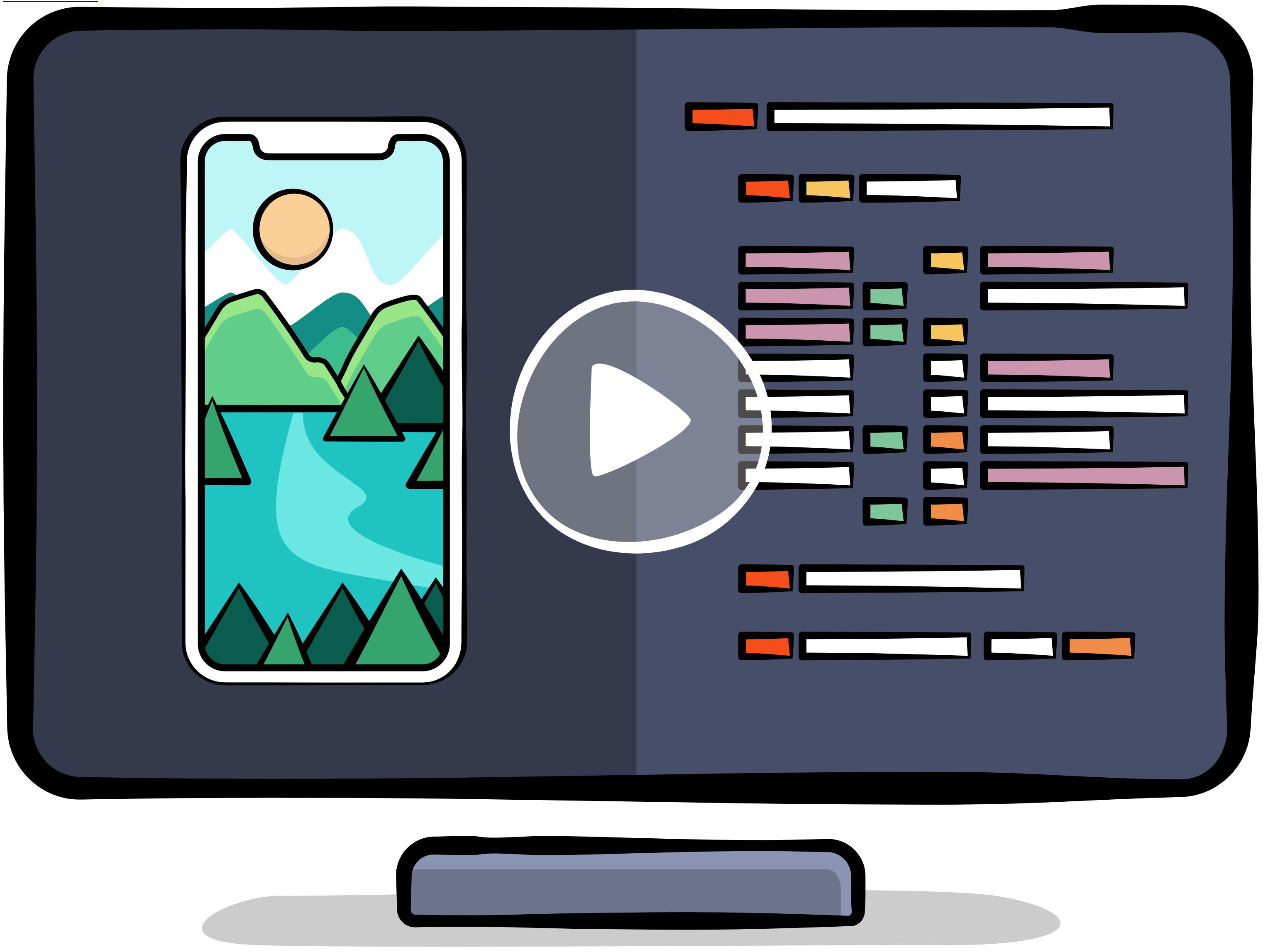
Download Materials

Mark tutorial complete

raywenderlich.com Subscription

Full access to the largest collection of Swift and iOS development tutorials anywhere!

[Learn more](#)



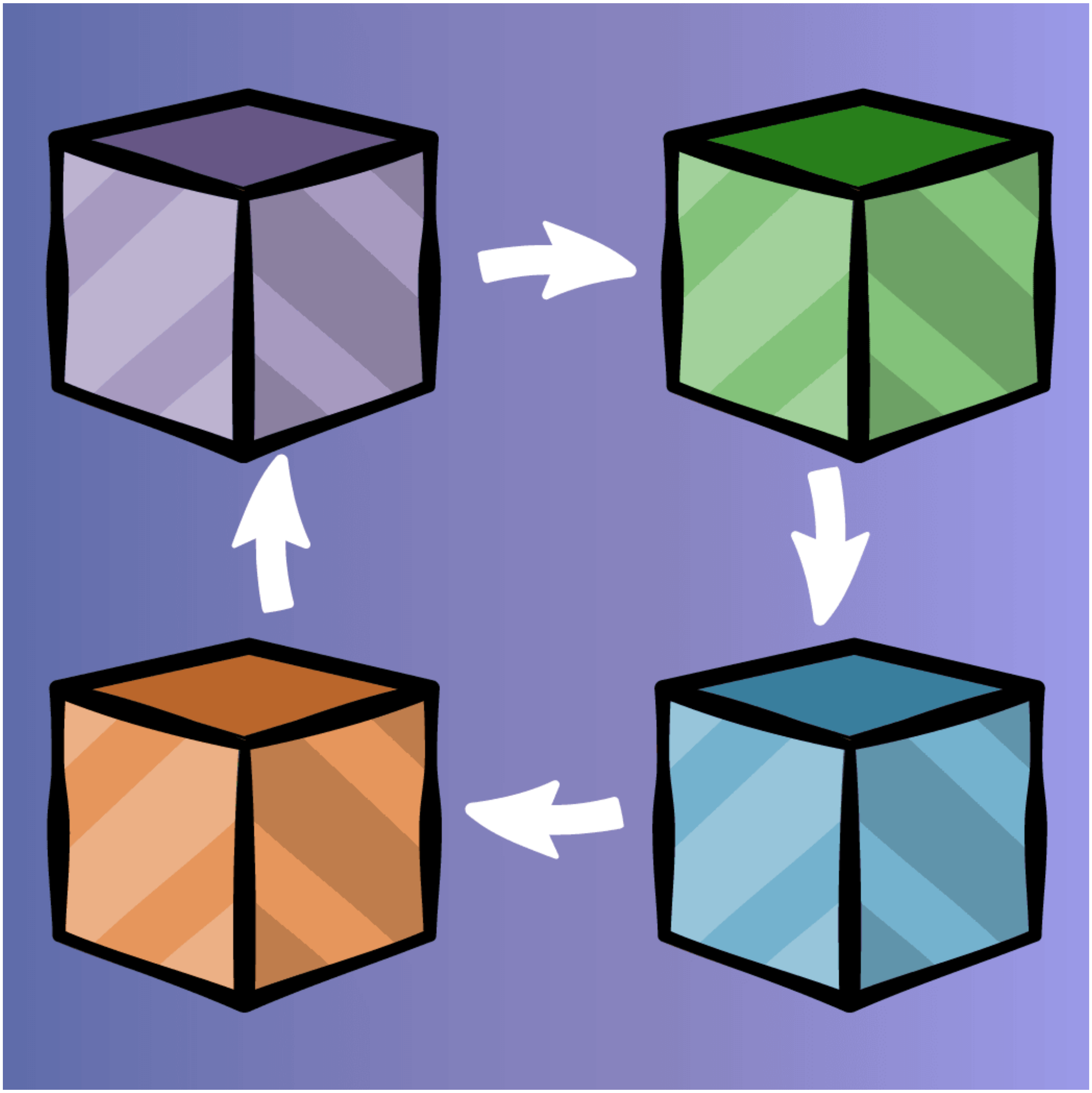
More like this

[New](#)

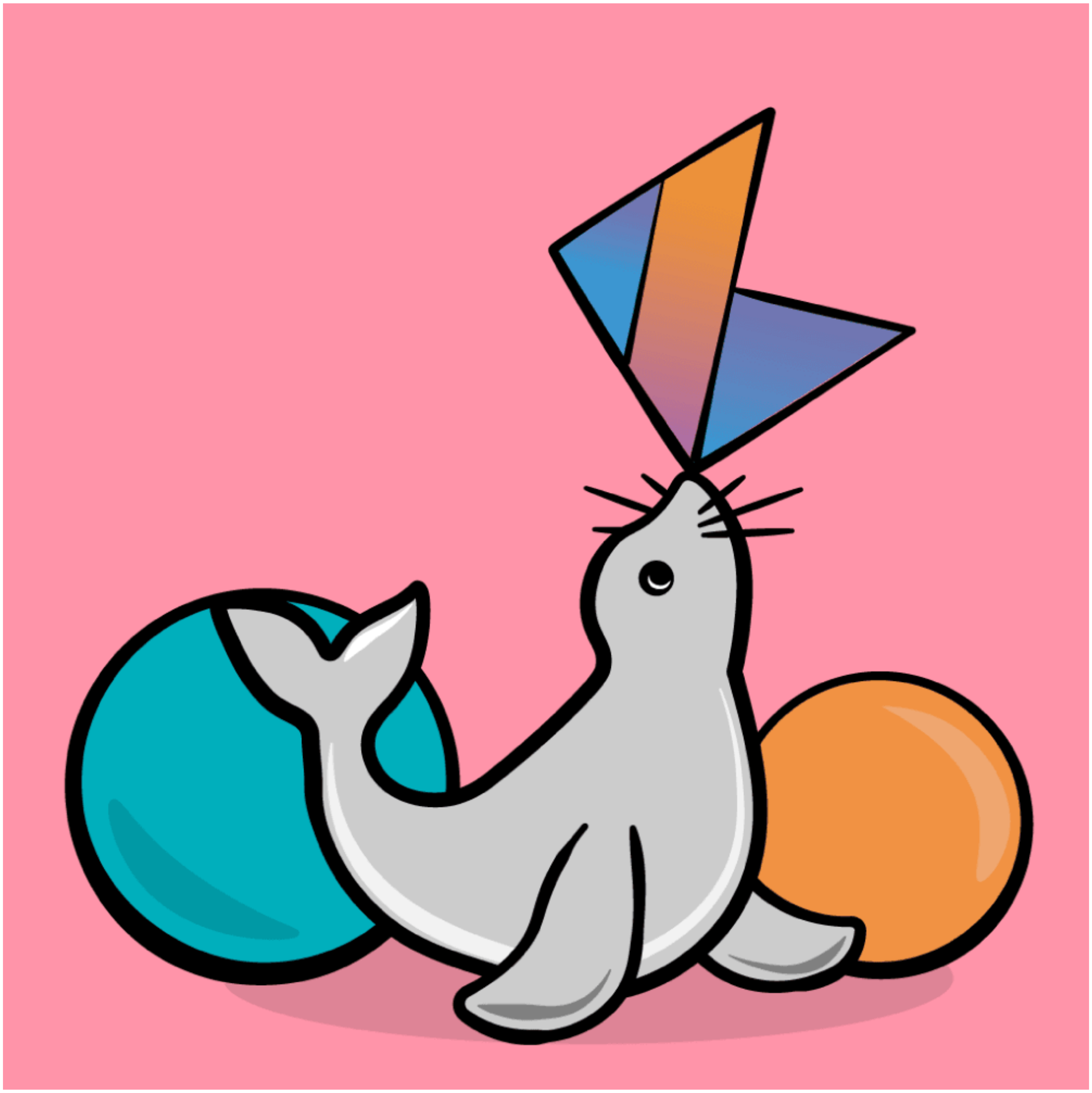
Requesting App Ratings and Reviews Tutorial for iOS Dec 17 2018 · Article · 20 mins Dec 17 2018 · 20 mins Article

Completed

New



[How to use the New Unity Prefab Workflow Dec 15 2018 · Article · 25 mins Dec 15 2018 · 25 mins Article Completed New]



Kotlin Sealed Classes Dec 12 2018 · Article · 10 mins Dec 12 2018 · 10 mins Article New Completed

Getting Started With PromiseKit Dec 12 2018 · Article · 30 mins Dec 12 2018 · 30 mins Article Completed

Contributors

Pietro Rea

iOS @UpsideTravel. Co-author of Core Data by Tutorials and iOS 9 by Tutorials.

Author

Cesare Rocchi

Cesare Rocchi runs Studio Magnolia, an interactive studio that creates compelling web and mobile applications. He blogs at ...

Tech Editor

Erik Kerber

Erik is an iOS guy from Minneapolis. Currently a Lead iOS developer at Target, and an independent consultant at Distilled Bits....

Fpe

Richard Critz

Richard is the iOS Team Lead for RayWenderlich.com. He has been doing software professionally for nearly 40 years, working on...

Team Lead

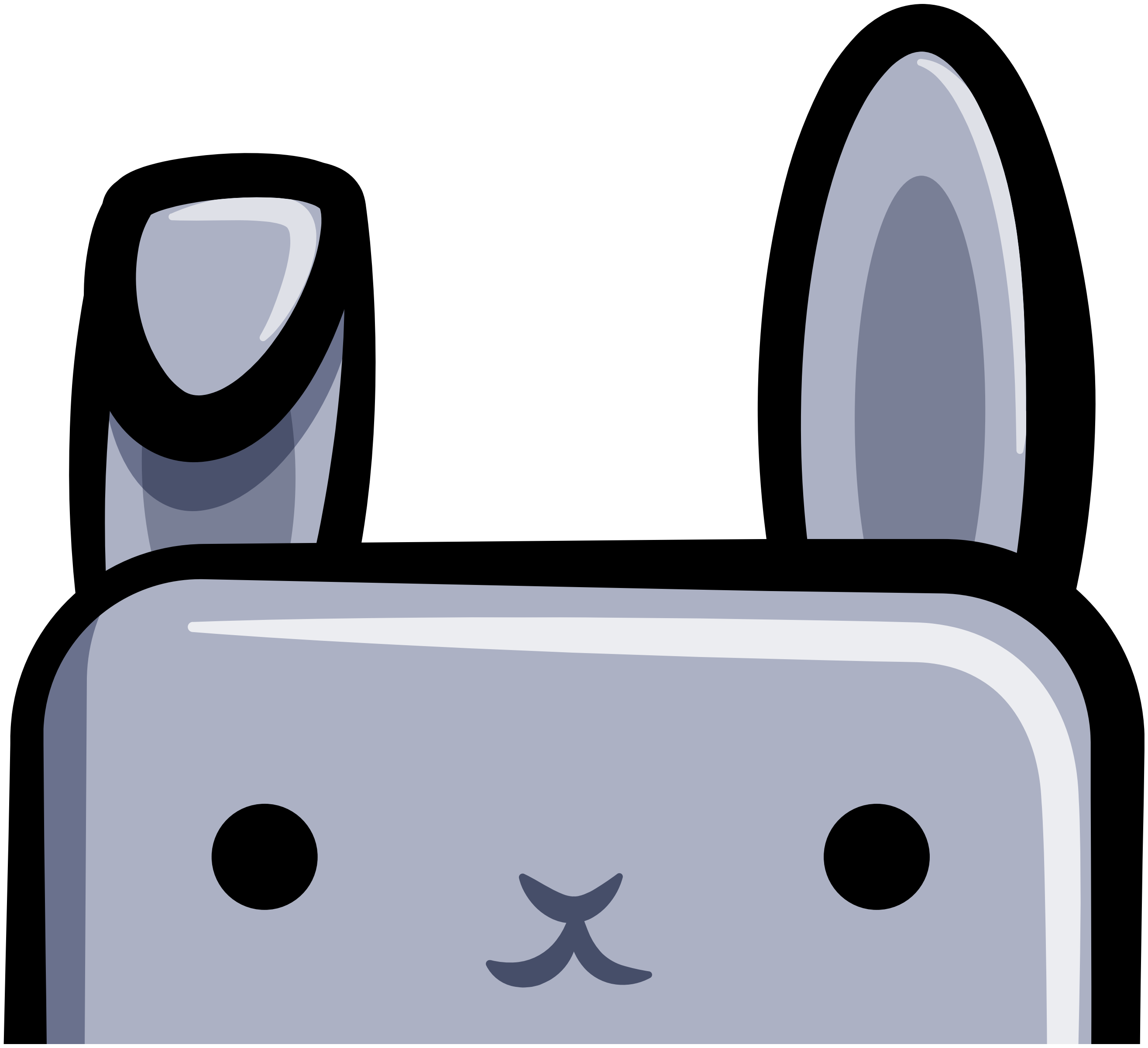
Comments

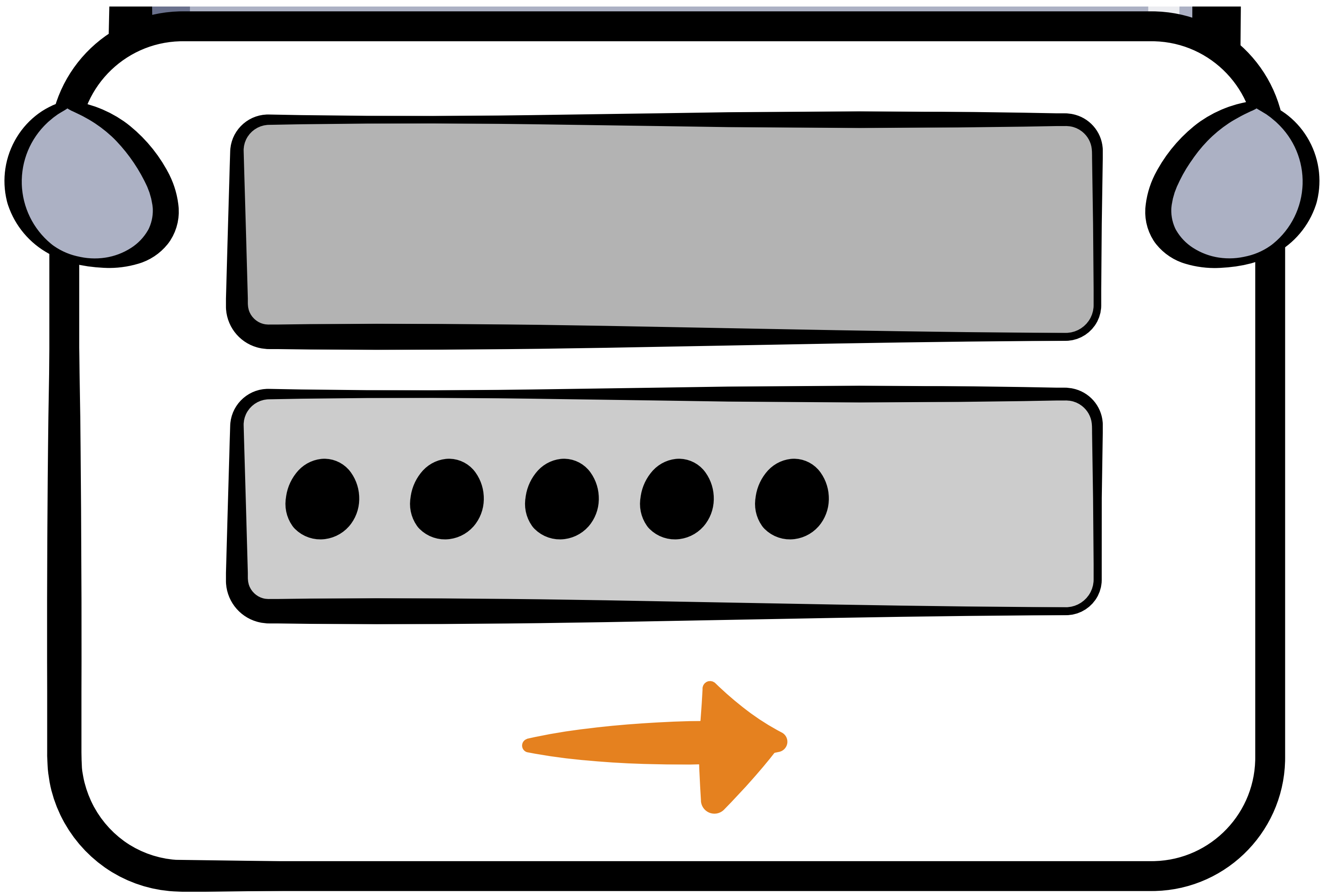
Show Comments.

Create your free learning account today!

With a free raywenderlich.com account, you can download source code from our tutorials, track your progress, personalize your learner profile, participate in open discussion forums and more!

Get Started





The largest and most up-to-date collection of development courses on iOS, Swift, Android, Kotlin, Unity, Unreal Engine and more.

-

-

Places

- Library
- Help
- About
- Newsletter
- Forums
- Podcast
- Store

raywenderlich.com Weekly

Get weekly digests of our tutorials and courses, and receive a free epic-length email course as a bonus!

→

stevewozniak@apple.com