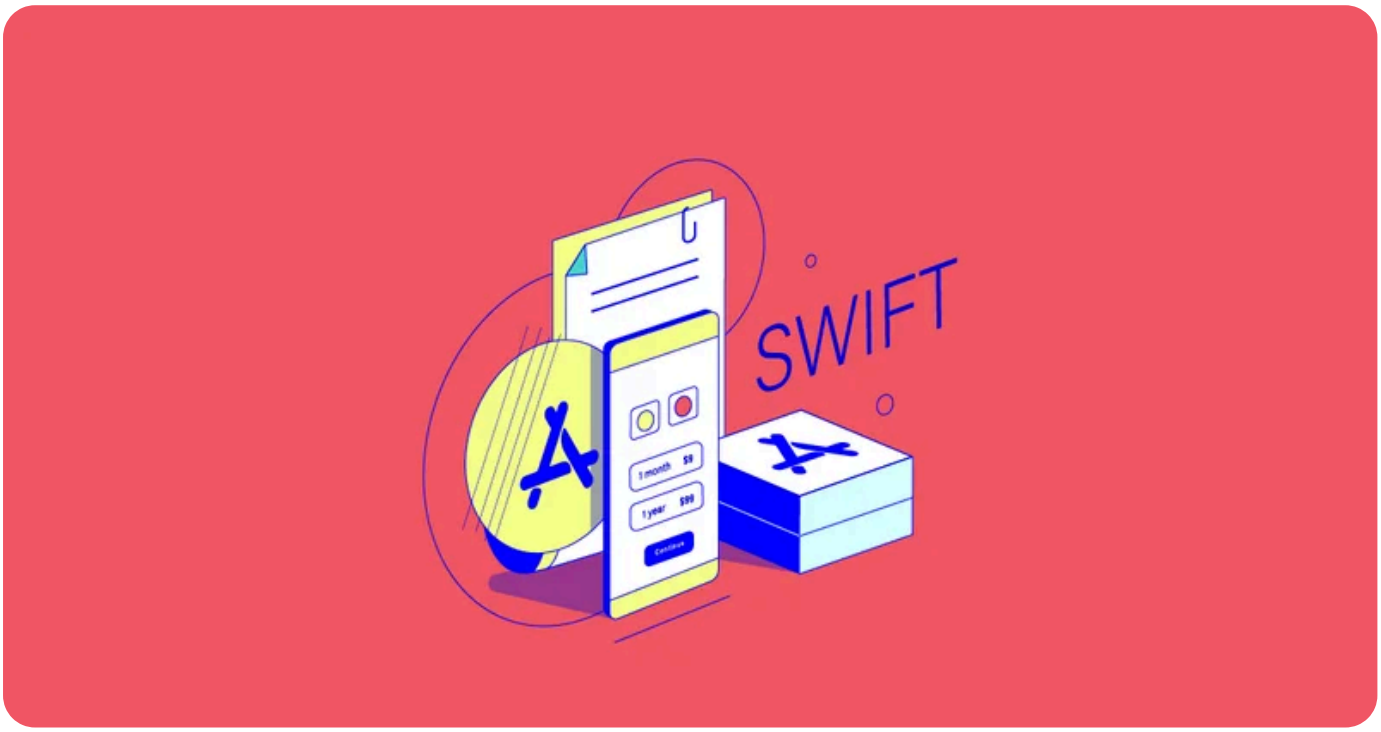
RevenueCat

ENGINEERING

# iOS In-App Subscription Tutorial with StoreKit 2 and Swift

A step-by-step guide to a working SwiftUI sample app with subscriptions.




Josh Holtz

Published **December 13, 2022**
Last updated **June 6, 2024**

Introduction

This StoreKit 2 tutorial is paired with code examples and a sample app that can be found at https://github.com/RevenueCat/storekit2-demo-app.

## Introduction

In-app purchases and subscriptions are one of the best ways for apps to make money on the App Store. StoreKit 2 is Apple's newly updated framework for in-app purchases, which enables

developers to add IAPs to their iOS, macOS, watchOS, and tvOS apps. Apple's documentation does a great job of giving a high-level explanation of how to use StoreKit, but doesn't go into the intricacies or provide a full working example.

This tutorial will cover the basic concepts, App Store Connect setup, how to add StoreKit 2, display, complete, and verify the purchase, as well as how to handle changes that happen outside the app (renewal, cancellation, billing issue, etc). It will also talk about the benefits and tradeoffs of having a server or not. It will feature Swift code samples and a downloadable, runnable sample app.

## Table of contents

## Terminology

**In-app purchases** (IAPs) are digital products that are purchased for use within an app. The purchase will most often take place directly within an app but sometimes can be initiated from the App Store.

Apple has four types of IAPs:

**Consumables** are products that can be purchased once, more than once, and in quantity. Examples of consumables could be "lives" or gems in games, boosts in a dating app to increase visibility, or tips for developers or creators.

**Non-consumables** are products that can only be purchased once and don't expire. Examples of non-consumables could be unlocking premium features in an app, getting extra brushes in a painting app, or obtaining cosmetic items in a game.

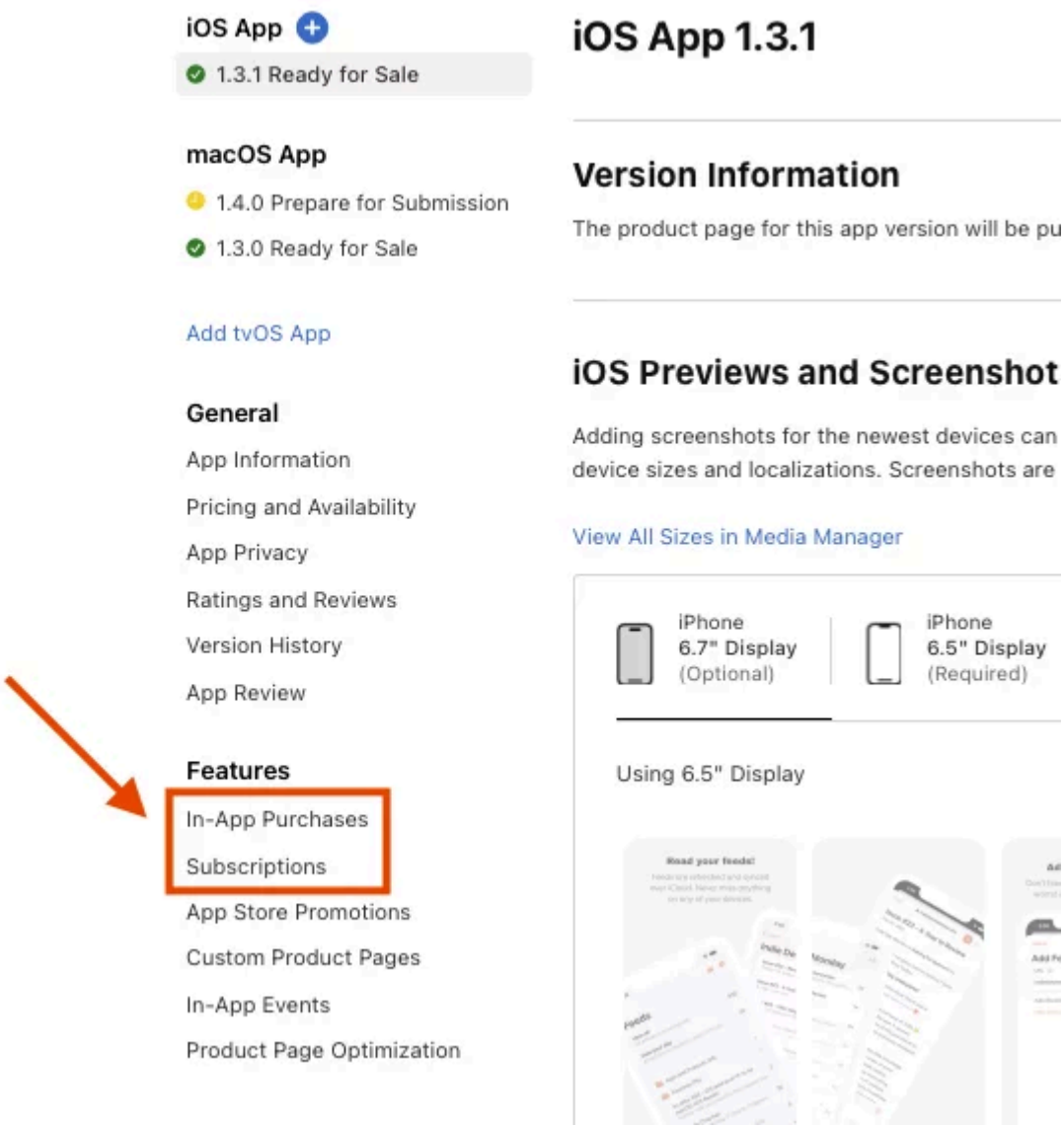
**Auto-renewable subscriptions** are products that can provide ongoing access to content, services, or premium features in an app. Customers are charged on a recurring basis until they cancel or billing issues occur. Examples of auto-renewable subscriptions could include access to a software service or content (like lessons in an education app).

**Non-renewing subscriptions** are products that provide access to content, services, and premium features in an app for a limited time without automatic renewal. This type of subscription requires the user to manually resubscribe to regain access to the service or content. An example of a non-renewing subscription could be a season pass to in-game content.

## Setting up App Store Connect

The first step to adding in-app purchases to an app is to create the products in App Store Connect, the developer dashboard used to put apps on the App Store. Developers can use App Store Connect to create the app listing, write the app description, upload screenshots, create in-app products, manage testers, and release new versions of the app to the App Store.

There are two different sections for creating and managing in-app purchases. The first section is “In-App Purchases” and the second is the “Subscriptions”. “In-App Purchases” is for consumables and non-consumables, while “Subscriptions” is for auto-renewable and non-renewing subscriptions. Subscriptions have been separated because their configuration is more complex than consumables and non-consumables.




App Store Connect dashboard pointing at the location on of “In-App Purchases” and “Subscriptions”

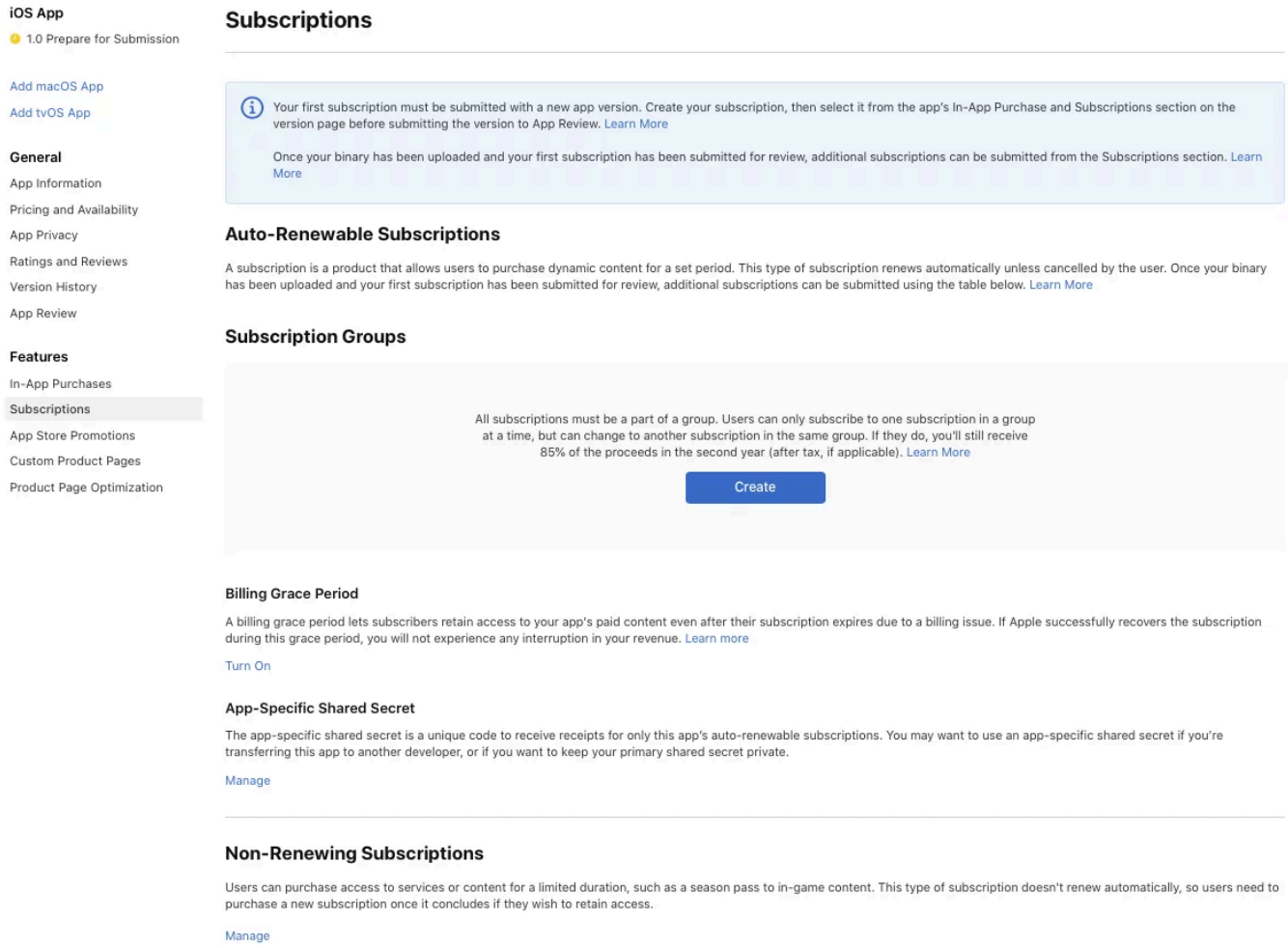
### App Store Connect requirements

There are a few administrative items that must be completed before your app can sell in-app purchases:

1. Setup bank information
2. Sign Paid App agreement

### Create subscriptions (auto-renewable and non-renewing)

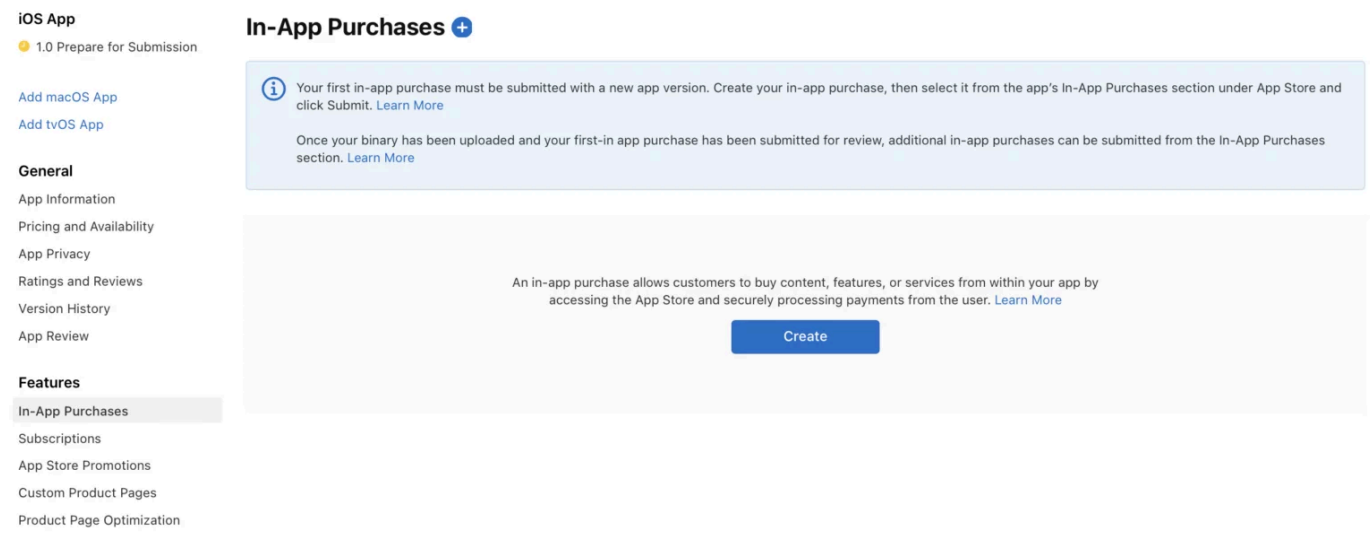
1. Go to “Subscriptions”
2. Create a “Subscription Group”
3. Add a localization for the “Subscription Group”
4. Create new subscription (reference name and product id)
5. Fill out all metadata (duration, price, localization, review information)




App Store Connect screen to add auto-renewable and non-renewing subscriptions

## Create consumables and non-consumables

1. Go to “In-app Purchases”
2. Create a new in-app purchase
3. Select the type (consumable or non-consumable) and set reference name and product id
4. Fill out all metadata (price, localization, review information)




App Store Connect screen to add consumable and non-consumable in-app purchases

## Setting up StoreKit configuration file

Setting up products in App Store Connect can be a lot of work: It may seem like a very large first step when starting the journey of in-app products and subscriptions. However, even though these steps are necessary for releasing an app with IAPs, you don't need the App Store to do local development! As of Xcode 13, the entire in-app purchase workflow can be done by using a StoreKit configuration file.

Using the StoreKit configuration has many more benefits than just deferring logging into App Store Connect. It can also be used for:

- Testing purchase flows in the simulator
- Testing purchase flows in unit and UI tests
- Testing locally when no network connection is available
- Debugging edge cases that are difficult to set up or reproduce in the sandbox environment
- Testing end-to-end transactions with failures, renewals, billing issues, promotional offers, and introductory offers

In fact, the example code and sample app paired with this tutorial use a StoreKit configuration file. The source code can be downloaded and run without the need to configure anything on App Store Connect.

There are some great WWDC videos that show all the powers of the StoreKit Configuration File:

WWDC20 – Introducing StoreKit Testing in Xcode

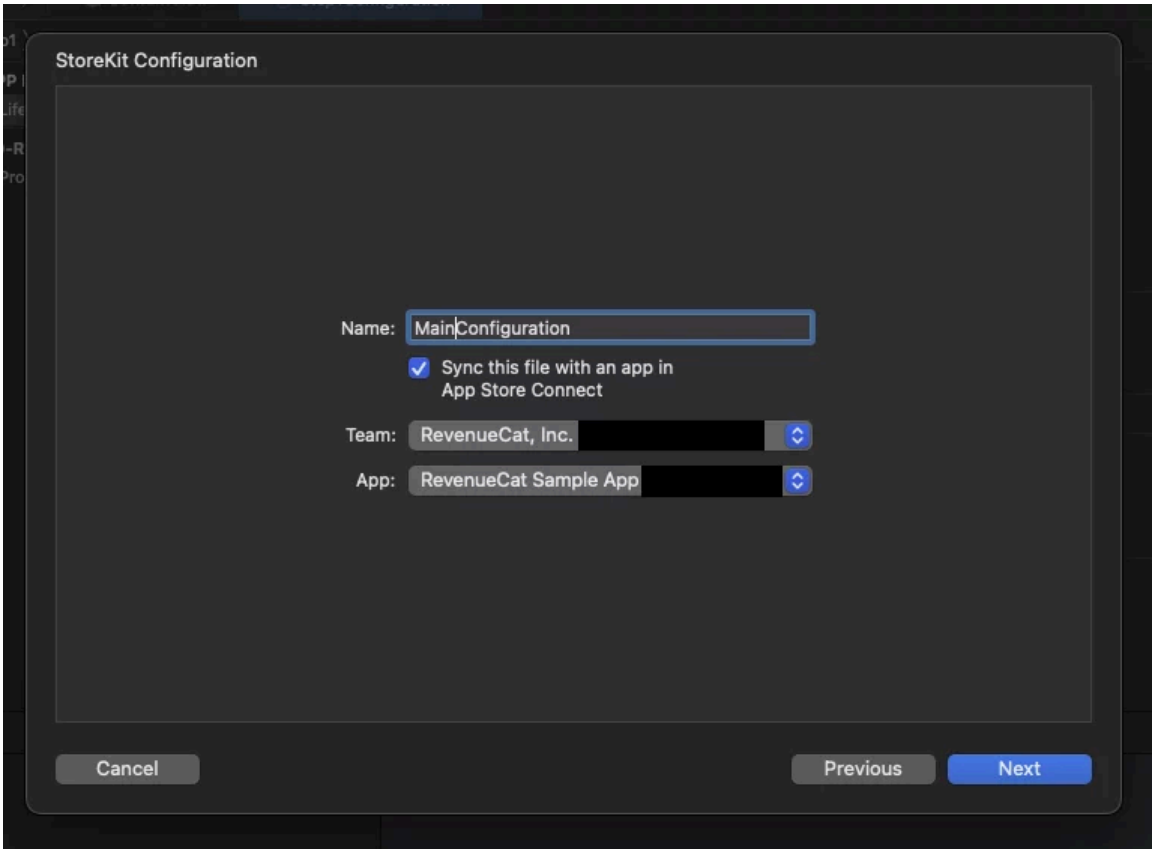
WWDC22 – What's new in StoreKit testing

We also have a blog post on StoreKit Testing Improvements in iOS 14 that does a great job of summarizing these WWDC videos.

The following is a quick overview of the steps involved in creating, configuring, and enabling a StoreKit configuration file.

### Create the file

1. Launch Xcode, then choose File > New > File.
2. Search for "storekit" in the Filter search field.
3. Select "StoreKit Configuration File"
4. Name it, check "Sync this file with an app in App Store Connect", and save it.

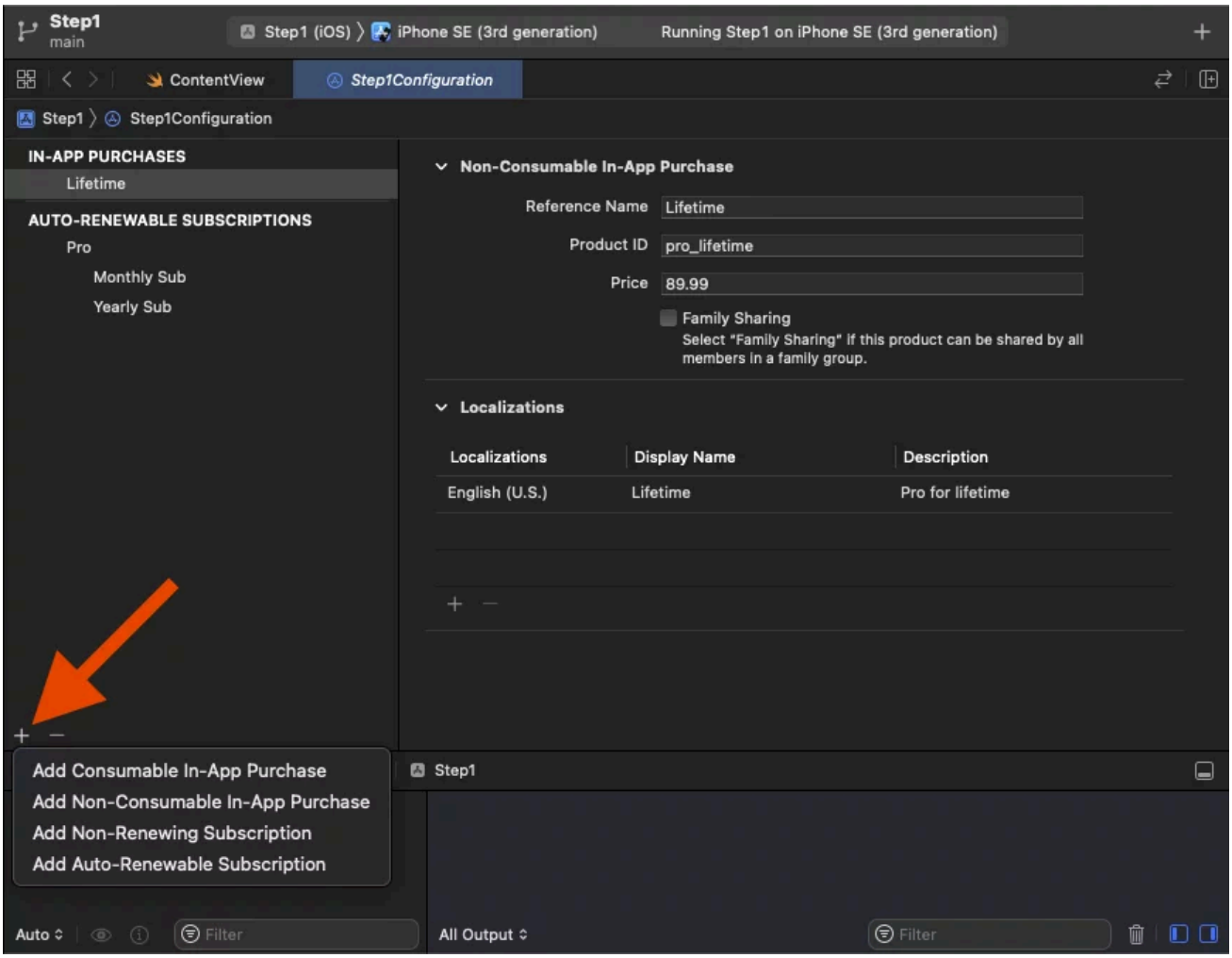



### Add products (optional)

Xcode 14 added the ability to sync this file with an app in App Store Connect. This will prevent you from manually having to add products to the StoreKit Configuration File, which is useful if you already have products defined in App Store Connect that you would like to mirror for local testing.

However, if you are using Xcode 13 yet or want to test with other product types or duration, you can still do that.

1. Click “+” in the bottom left corner in the StoreKit Configuration File editor in Xcode.
2. Select a type of in-app purchase.
3. Fill out the required fields:
    1. Reference name
    2. Product ID
    3. Price
    4. At least one localization



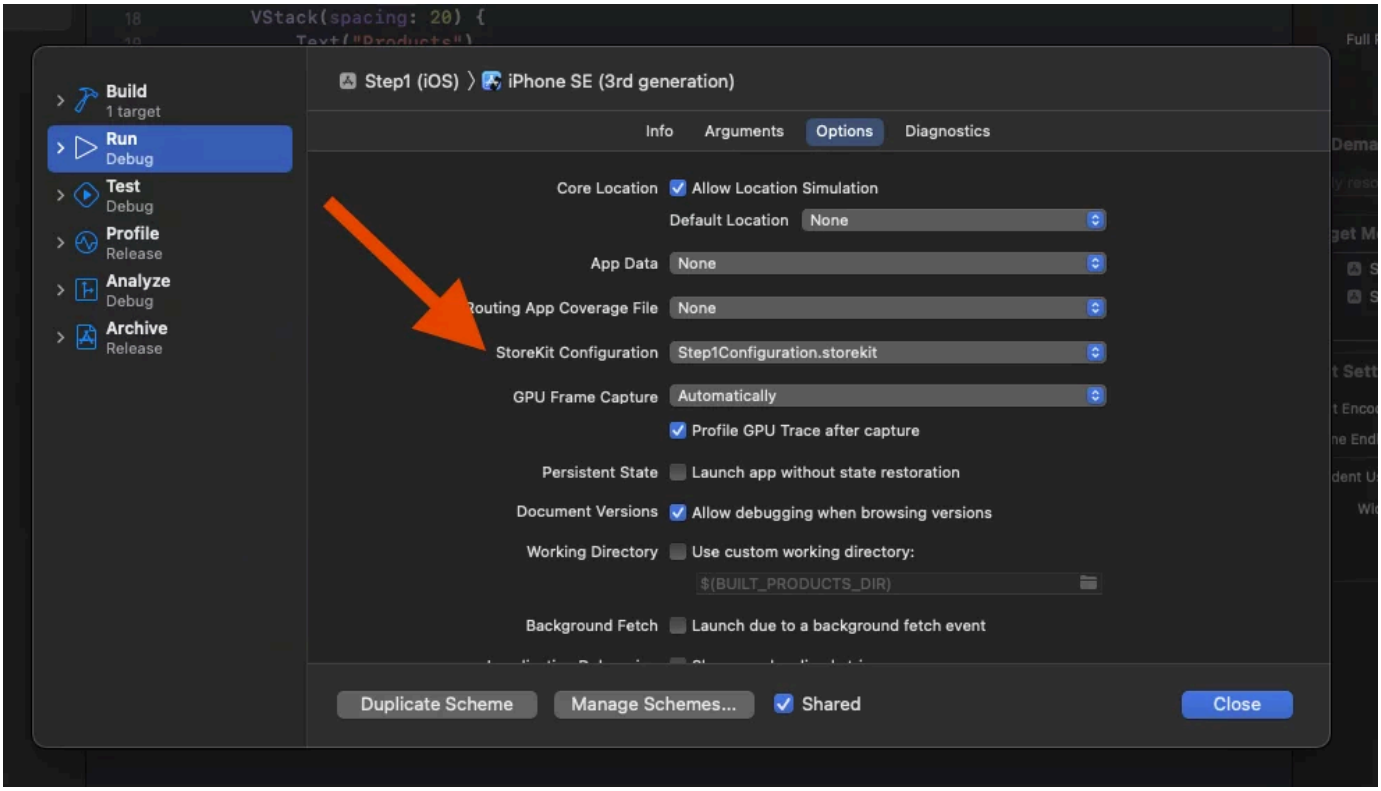

### Enable the StoreKit Configuration File

Simply creating StoreKit Configuration File isn’t enough to use it. The StoreKit Configuration File needs to be selected in an Xcode scheme.

1. Click scheme and select “Edit scheme.”

2. Go to Run > Options.

3. Select a value for “StoreKit Configuration.”




Even though using the StoreKit configuration file is the easiest way to test IAPs, there are times when it is necessary to test the sandbox environment on a device. It can be a hassle to keep changing the scheme between these two to edit, so we recommend duplicating the scheme specifically for the StoreKit configuration file and naming it something like “YourApp (SK Config).”

## Implementing on-device subscriptions with StoreKit 2 in Swift

Whether they’re created with App Store Connect or a StoreKit configuration file, ensuring that there are products for the app to fetch is the first step before trying to interact with any of the StoreKit APIs with Swift.

This section is a step-by-step guide for using StoreKit 2 to write the code to:

List products

Purchase products

Unlock features for active subscriptions and lifetime purchases

Handle renewals, cancellations, and billing errors

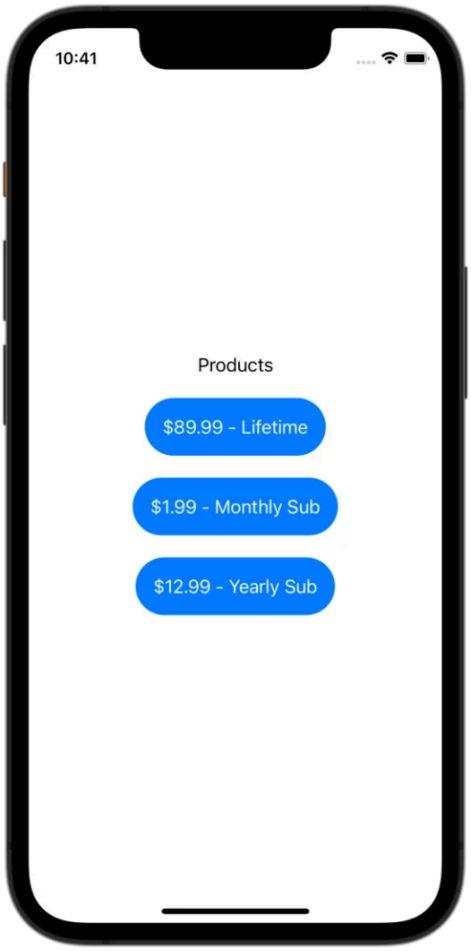
Validate receipts

These steps will be implemented in a way that requires no external server. Performing in-app purchase logic without a backend was difficult and insecure with StoreKit 1, but Apple has made some big improvements to StoreKit 2 to make this possible and secure. Processing purchases in your backend requires more work but provides many benefits, which we'll explore later.

Let's get started!

## Step 1: Listing products

We start by showing the user a paywall of all purchasable in-app products, which appear as buttons that allow the user to purchase the products when tapped. In this example, our app will show a monthly subscription, a yearly subscription, and a lifetime unlock purchase. Here's what will be implemented at the end of the first step:




The example paywall

Fetching products from StoreKit 2 only takes a few lines of code:

```
import StoreKit

let productIds = ["pro_monthly", "pro_yearly", "pro_lifetime"]
let products = try await Product.products(for: productIds)
```

The snippet above imports StoreKit, defines an array of strings of product identifiers to show in the paywall, and then fetches those products. The product identifiers should match the products defined in a StoreKit Configuration File or App Store Connect. The end result is an array of `Product` objects. A `Product` object contains all the information needed to show in the buttons of the paywall. These `Product` objects will eventually be used to make a purchase when the button is tapped.

The product identifiers are hardcoded in this example, but they shouldn't be hardcoded in a production app. Hardcoded product identifiers make it difficult to swap out different products by requiring an app update. Not all users have automatic updates on, which means that some product identifiers could be purchased for longer than intended. It is best to serve the list of available products to be purchased from a remote source.

Now it's time to show those products in the view. Below is a SwiftUI view that will store the products in a state variable.

```
import SwiftUI
import StoreKit

struct ContentView: View {
    let productIds = ["pro_monthly", "pro_yearly", "pro_lifetime"]

    @State
    private var products: [Product] = []

    var body: some View {
        VStack(spacing: 20) {
            Text("Products")
            ForEach(self.products) { product in
                Button {
                    // Don't do anything yet
                } label: {
                    Text("\(product.displayPrice) - \(product.displayName)")
```

```
                }
            }
        }.task {
            do {
                try await self.loadProducts()
            } catch {
                print(error)
            }
        }
    }

    private func loadProducts() async throws {
        self.products = try await Product.products(for: productIds)
    }
}
```

In this snippet, the StoreKit 2 method for fetching products is moved into a new `loadProducts()` function. This function is then called when our view appears by using `.task()`. The products are iterated over within a `ForEach` loop that will display a `Button` for each product. This button currently does not do anything when tapped.

The full implementation up to this step can be found here.

## Step 2: Purchasing products

Now that the user can see all of the products on the paywall, the next step is to allow the purchase of the product when a button is tapped.

Initiating a purchase with a `Product` is as simple as calling the `purchase()` function on the product.

```
private func purchase(_ product: Product) async throws {
    let result = try await product.purchase()
}
```

If this method throws (either `Product.PurchaseError` or `StoreKitError`), the purchase didn't go through. **However, the absence of an error still does not indicate that the purchase was successful.**

The result of `purchase()` is a `Product.PurchaseResult` enum. A `PurchaseResult` looks like the following:

```
public enum PurchaseResult {
    case success(VerificationResult<Transaction>)
    case userCancelled
    case pending
}

public enum VerificationResult<SignedType> {
    case unverified(SignedType, VerificationResult<SignedType>.VerificationError)
    case verified(SignedType)
}
```

The snippet below shows how to update the app's `purchase` function to check for all the possible results from purchasing a product.

```
private func purchase(_ product: Product) async throws {
    let result = try await product.purchase()

    switch result {
    case let .success(.verified(transaction)):
        // Successful purhcase
        await transaction.finish()
    case let .success(.unverified(_, error)):
        // Successful purchase but transaction/receipt can't be verified
        // Could be a jailbroken phone
        break
    case .pending:
        // Transaction waiting on SCA (Strong Customer Authentication) or
        // approval from Ask to Buy
        break
    case .userCancelled:
        // ^^^
        break
    @unknown default:
        break
    }
}
```

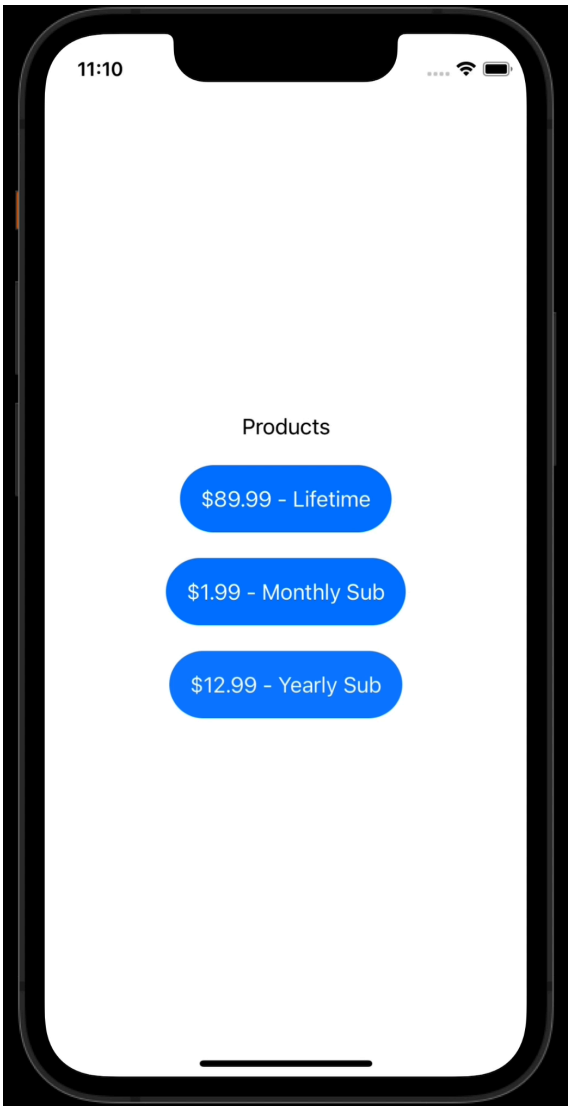
This table explains all of the different result types and what they mean:

| Case | Description |
| --- | --- |
| Success – verified | The in-app product was successfully purchased. |
| Success – unverified | The purchase was a success but failed StoreKit automatic verification checks. This could be due to a jailbroken device, but the StoreKit documentation is currently unclear on this. |
| Pending | This is caused by either Strong Customer Authentication (SCA) or Ask to Buy. SCA is an additional factor of authentication or approval by a financial authority before this purchase is made. This might be done through an app or through a text message. After approval, the transaction will be updated. Ask to Buy is a feature enabling a child to ask to purchase something from a parent or guardian. The purchase will be held in a pending state until the parent or guardian approves or denies the purchase. |
| User Canceled | This usually does not need to get handled as an error state, but it is sometimes good to know. |
| Error | This result is either `Product.PurchaseError` or `StoreKitError` and could be due to not having an internet connection, App Store being down, payment errors, among other causes. This . |

The last required piece for making a purchase is to call the `purchase(_ product: Product)` function when the button is tapped.

```
ForEach(self.products) { (product) in
    Button {
        Task {
            do {
                try await self.purchase(product)
            } catch {
                print(error)
            }
        }
    } label: {
        Text("\(product.displayPrice) - \(product.displayName)")
```

```
12        }
13    }
```




The full implementation up to this step can be seen here.

## Step 3: Preparing for unlocking in-app features

The user can now see all the available in-app products (two subscriptions and a lifetime purchase in this example) and purchase a product when tapping a button. However, the implementation is still incomplete: Features might need to be unlocked, or content might need to be downloaded when a purchase is made, or a subscription is active.

Unlocking content and features for users after a purchase is where things start to become more complex. It would be ideal if we only needed to add feature unlocking logic after a successful purchase was made, but that is not enough. There are multiple purchase flows aside from successful purchase, and the previous step alluded to two of them with Strong Customer

Authentication and Ask to Buy. In-app purchases can also be made outside of the app by directly purchasing them through the App Store. Successful purchases can happen at any time, and our app needs to be prepared to handle every situation.

Before starting to handle all these cases, we need to clean up the existing implementation. Currently, all the in-app purchasing logic lives within the SwiftUI view. This was okay when trying to get an end-to-end purchase flow working but does not scale well as the app grows. All the IAP logic should be moved out of the view and into a reusable component. This can be done many ways, so each app will do it differently, but in this step we will move the IAP logic into a new `PurchaseManager` object. The `PurchaseManager` object will initially be responsible for loading the in-app products and purchasing a product, but more functionality will be added soon.

```
import Foundation
import StoreKit

@MainActor
class PurchaseManager: ObservableObject {

    private let productIds = ["pro_monthly", "pro_yearly", "pro_lifetime"]

    @Published
    private(set) var products: [Product] = []
    private var productsLoaded = false

    func loadProducts() async throws {
        guard !self.productsLoaded else { return }
        self.products = try await Product.products(for: productIds)
        self.productsLoaded = true
    }

    func purchase(_ product: Product) async throws {
        let result = try await product.purchase()

        switch result {
        case let .success(.verified(transaction)):
            // Successful purhcase
            await transaction.finish()
        case let .success(.unverified(_, error)):
            // Successful purchase but transaction/receipt can't be verified
            // Could be a jailbroken phone
```

```
            break
        case .pending:
            // Transaction waiting on SCA (Strong Customer Authentication) or
            // approval from Ask to Buy
            break
        case .userCancelled:
            // ^^^
            break
        @unknown default:
            break
        }
    }
}
```

The `loadProducts()` and `purchase()` functions were moved into `PurchaseManager`, and the `ContentView` will use this `PurchaseManager` object. However, `ContentView` will not be the owner of the `PurchaseManager`. It will be created in `App` and get passed down into `ContentView` as an environment object. This approach allows other SwiftUI views to easily gain access to the same `PurchaseManager` object.

```
struct YourApp: App {
    @StateObject
    private var purchaseManager = PurchaseManager()

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(purchaseManager)
        }
    }
}
```

Because the `PurchaseManager` object is an `ObservableObject`, the SwiftUI view will automatically update when its properties change. Right now, the only property that `PurchaseManager` has is `products`.

```
struct ContentView: View {
    @EnvironmentObject
    private var purchaseManager: PurchaseManager

```

```
    var body: some View {
        VStack(spacing: 20) {
            Text("Products")
            ForEach(purchaseManager.products) { product in
                Button {
                    Task {
                        do {
                            try await purchaseManager.purchase(product)
                        } catch {
                            print(error)
                        }
                    }
                } label: {
                    Text("\(product.displayPrice) - \(product.displayName)")
                        .foregroundColor(.white)
                        .padding()
                        .background(.blue)
                        .clipShape(Capsule())
                }
            }
        }.task {
            Task {
                do {
                    try await purchaseManager.loadProducts()
                } catch {
                    print(error)
                }
            }
        }
    }
}
```

The app will run exactly as it did in Step 2, but now the code is more manageable when adding support for more in-app purchasing use cases.

The full implementation up to this step can be seen here.

## Step 4: Unlocking in-app features

`PurchaseManager` is now the perfect place to handle the logic for determining whether in-app features should be unlocked. This will all be built around the StoreKit 2's new `Transaction.currentEntitlements` function.

```
for await result in Transaction.currentEntitlements {
    // Do something with transaction
}

```

The name `currentEntitlements` might sound a little foreign to the rest of StoreKit 2's naming, but `Transaction.currentEntitlements` simply returns an array of transactions that are active. The **documentation** for `currentEntitlements` explains it as follows:

> A transaction for each non-consumable in-app purchase
>
> The latest transaction for each active auto-renewable subscription
>
> The latest transaction for each non-renewing subscription
>
> A transaction for each **unfinished** consumable in-app purchase

`PurchaseManager` will iterate over these transactions and store the product identifiers into a `Set` (an array that can't contain duplicate items) called `purchasedProductIDs`. These active product identifiers can now be used to unlock features and content in the app for users that have purchased them.

```
class PurchaseManager: ObservableObject {

    ...

    @Published
    private(set) var purchasedProductIDs = Set<String>()

    var hasUnlockedPro: Bool {
       return !self.purchasedProductIDs.isEmpty
    }

    func updatePurchasedProducts() async {
        for await result in Transaction.currentEntitlements {
            guard case .verified(let transaction) = result else {
                continue
            }

            if transaction.revocationDate == nil {
                self.purchasedProductIDs.insert(transaction.productID)
            } else {
```

```
                self.purchasedProductIDs.remove(transaction.productID)
            }
        }
    }
}

```

This new `updatePurchasedProducts` function needs to be called on application start, after a purchase is made, and when transactions are updated to ensure that `purchasedProductIDs` (and `hasUnlockedPro`) return the correct value that the user is expecting.

First, the `App` needs to be updated to call `updatePurchasedProducts()` on application start. This will initialize the purchased products array with the current entitlements.

```
struct YourApp: App {
    @StateObject
    private var purchaseManager = PurchaseManager()

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(purchaseManager)
                .task {
                    await purchaseManager.updatePurchasedProducts()
                }
        }
    }
}

```

Next, the `purchase()` function needs to call `updatePurchasedProducts()` after a successfully verified purchase. This will update the purchased products array with the newly purchased product.

```
func purchase(_ product: Product) async throws {
    let result = try await product.purchase()

    switch result {
    case let .success(.verified(transaction)):
        await transaction.finish()
```

```
            await self.updatePurchasedProducts()
        case let .success(.unverified(_, error)):
            break
        case .pending:
            break
        case .userCancelled:
            break
        @unknown default:
            break
        }
    }

```

The last missing piece is listening for new transactions created outside the app. These transactions could be subscriptions that have been canceled, renewed, or revoked due to billing issues, but they could also be new purchases that happened on another device that should unlock content on this one. This monitoring can be done by listening for changes on the `Transaction.updates` async sequence:

```
@MainActor
class PurchaseManager: ObservableObject {

    ...

    private var updates: Task<Void, Never>? = nil

    init() {
        updates = observeTransactionUpdates()
    }

    deinit {
        updates?.cancel()
    }

    ...

    private func observeTransactionUpdates() -> Task<Void, Never> {
        Task(priority: .background) { [unowned self] in
            for await verificationResult in Transaction.updates {
                // Using verificationResult directly would be better
                // but this way works for this tutorial
```

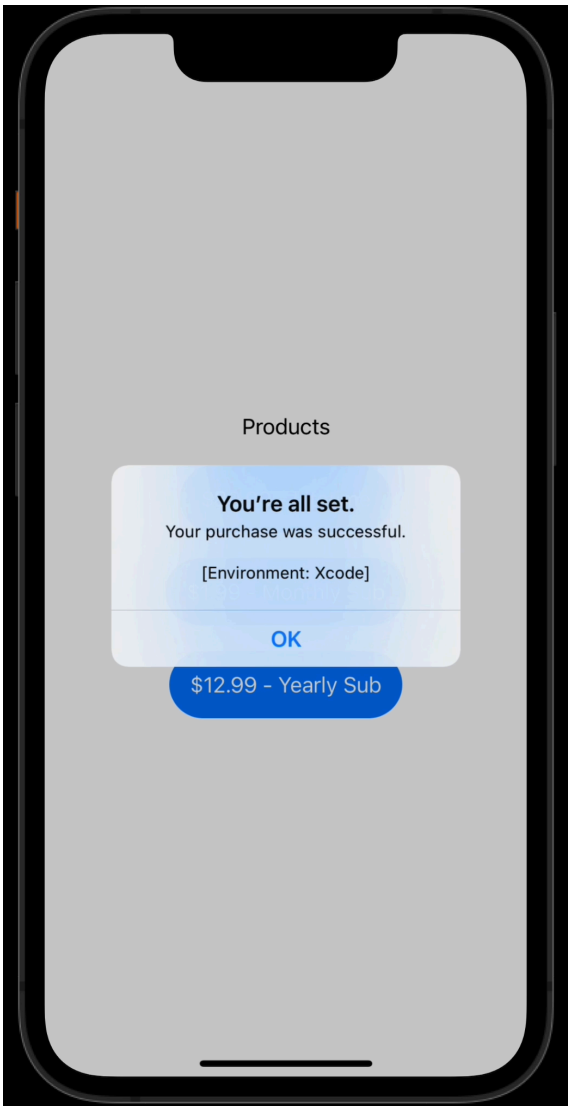
```
                await self.updatePurchasedProducts()
            }
        }
    }
}

```

With all the changes in place for `PurchaseManager`, the SwiftUI view only needs a few new lines of code to remove the paywall when a product has been purchased. This is done by adding an if statement checking `purchaseManager.hasUnlockedPro`. Because `PurchaseManager` is an `ObservableObject`, and the SwiftUI view will know to automatically refresh when its properties are changed.

```
var body: some View {
    VStack(spacing: 20) {
        if purchaseManager.hasUnlockedPro {
            Text("Thank you for purchasing pro!")
        } else {
            Text("Products")
            ForEach(purchaseManager.products) { (product) in
                Button {
                    Task {
                        do {
                            try await purchaseManager.purchase(product)
                        } catch {
                            print(error)
                        }
                    }
                } label: {
                    Text("\(product.displayPrice) - \(product.displayName)")
                        .foregroundColor(.white)
                        .padding()
                        .background(.blue)
                        .clipShape(Capsule())
                }
            }
        }
    }
}

```




The full implementation up to this step can be seen here.

## Step 5: Handling purchased products when offline

The previous step introduced `Transaction.currentEntitlements`, which was used to iterate over a sequence of the user's purchased products (non-consumables, active subscriptions, or unfinished consumables). To ensure that the user gets the expected app behavior for each subscription status, this function will fetch the latest transactions if there is internet access. If the user does not have an internet connection — Wi-Fi is down, airplane mode is on, etc. — `Transaction.currentEntitlements` will return locally cached data. Transactions are also pushed to the device when it's online, which could allow the app to have the most up-to-date transactions when the user occasionally goes offline.

StoreKit 2 did a really great job with this implementation. Developers don't need to worry about designing any logic or custom caching to make their StoreKit 2 enabled apps work offline. Fetching purchased products with `Transaction.currentEntitlements` works the same way whether online or offline.

## Step 6: Restoring purchases

As in the previous step, nothing needs to be done to restore purchases: StoreKit will automatically keep in-app subscription status and transaction history up to date through `Transaction.currentEntitlements` and `Transaction.all`. By using these functions, there is no technical reason for a user to manually attempt to restore transactions. However, it is still a best practice to have a "Restore Purchases" button in an app:

1. The App Store Review Guidelines, section 3.1.1, require a restore mechanism be present in an app for any restorable in-app purchases. Even though this doesn't explicitly state that a "Restore Purchases" button is necessary, and it could be debated that StoreKit's default behavior keeps all transactions in sync, having a "Restore Purchases" button easily satisfies any uncertainty in this guideline.
2. Even though this restore purchase functionality shouldn't be needed, it is always ideal to give users a sense of control. Users may wonder whether purchases are up to date or not, and this button allows them to make sure that they are.

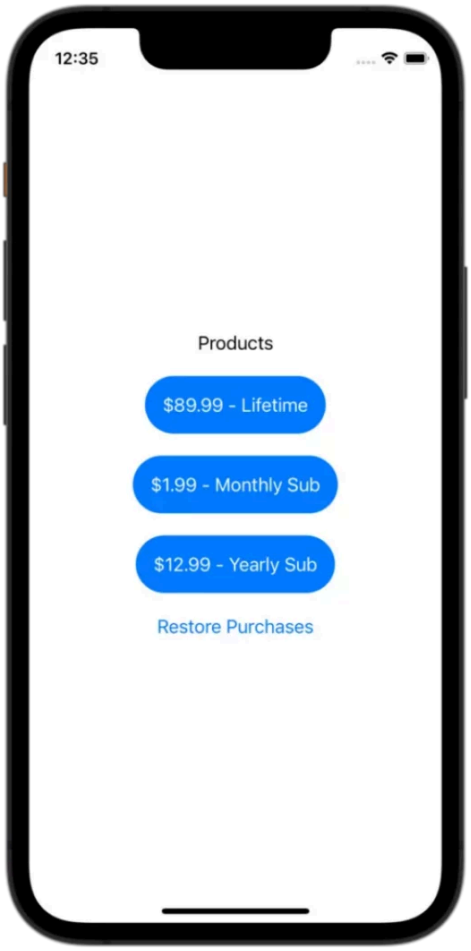
Adding in "Restore Purchases" functionality is trivial thanks to `AppStore.sync()`. According to its documentation, `AppStore.sync()` should be used in a "Restore Purchases" button and should rarely be needed, but is nice to have for a user who suspects that something is wrong with transactions or subscription status. There is a WWDC22 session on implementing proactive in-app purchase restore that goes deeper into this topic.

In rare cases, when a user suspects that the app isn't showing all the transactions, calling `AppStore.sync()`, which forces the app to obtain transaction information and subscription status from the App Store.

```
1   Each(purchaseManager.products) { product in
2       Button {
3           Task {
4               do {
5                   try await purchaseManager.purchase(product)
6               } catch {
7                   print(error)
8               }
9           }
10      } label: {
11          Text("\(product.displayPrice) - \(product.displayName)")
12      }
13  }
```

```
14
15  Button {
16      Task {
17          do {
18              try await AppStore.sync()
19          } catch {
20              print(error)
21          }
22      }
23  } label: {
24      Text("Restore Purchases")
25  }
26
```

A button calling `AppStore.sync()` will be placed on the paywall under the list of available products to purchase. In the unusual case where the user has purchased a product, but the paywall is still showing, `AppStore.sync()` will update the transactions, the paywall will disappear, and the purchased in-app content will be available for the user to use.




The full implementation up to this step can be seen here.

## Step 7: Sharing active purchases with extensions

iOS apps often include more than *just* the main app. Apps can be made up of Widget Extensions, Intent Extensions, Watch Apps, and much more. These extensions most likely run in separate contexts from the main app, but there is a good chance that subscription statuses could unlock functionality in these extensions. Luckily, again, StoreKit 2 makes this easy for most extensions.

`Transaction.currentEntitlements` can be used in extensions the same way it was used in the previous steps. This works for extensions like Widgets and Intents. However, an iOS app with a companion watchOS app will not work even though `Transaction.currentEntitlements` can be executed in it. A companion watch app does not stay updated with the same transaction history as its iOS app because they are separate platforms.

### Extensions

As mentioned earlier, `Transaction.currentEntitlements` will fetch locally cached transactions but can also get the latest transactions if the device is online. With app extensions having very limited runtime or limited capabilities, the behavior of `Transaction.currentEntitlements` might be too time consuming to repeatedly call for these extensions.

The best practice here is to have app extensions look at a boolean flag if certain products were purchased. This can be done by storing the shared information in a `UserDefaults` instance that is shared between apps and app extensions with app groups.

There will be a new `EntitlementsManager` class that will have the sole responsibility of storing the unlocked feature state that happens when purchasing a product. The `PurchaseManager` will update `EntitlementsManager` after calling `Transaction.currentEntitlements` from within the already existing `updatePurchasedProducts()` function.

```
1  import SwiftUI
2
3  class EntitlementManager: ObservableObject {
4      static let userDefaults = UserDefaults(suiteName: "group.your.app")!
5
6      @AppStorage("hasPro", store: userDefaults)
7      var hasPro: Bool = false
```

```
}

```

`EntitlementManager` is another `ObservableObject`, so the SwiftUI view can observe the changes and update whenever anything changes in it. SwiftUI comes with a very nice property wrapper called `@AppStorage` which saves its value to `UserDefaults(suiteName: "group.your.app")` and will also act as an `@Published` variable, which will redraw your SwiftUI views when updated.

The next step is to give `PurchaseManager` an instance of `EntitlementManager` to update in `updatePurchasedProducts()`.

```
class PurchaseManager: ObservableObject {

    ...

    private let entitlementManager: EntitlementManager

    init(entitlementManager: EntitlementManager) {
        self.entitlementManager = entitlementManager
    }

    ...

    func updatePurchasedProducts() async {
        for await result in Transaction.currentEntitlements {
            guard case .verified(let transaction) = result else {
                continue
            }

            if transaction.revocationDate == nil {
                self.purchasedProductIDs.insert(transaction.productID)
            } else {
                self.purchasedProductIDs.remove(transaction.productID)
            }
        }

        self.entitlementManager.hasPro = !self.purchasedProductIDs.isEmpty
    }
}

```

The last rework is in `App` with the initialization of `PurchaseManager`. Previously, `PurchaseManager` could be created as a `StateObject` with `@StateObject var purchaseManager = PurchaseManager()`. Now we'll have two `StateObject` variables where one has a dependency on the other so the initialization code has to be more verbose.

```
struct YourApp: App {
    @StateObject
    private var entitlementManager: EntitlementManager

    @StateObject
    private var purchaseManager: PurchaseManager

    init() {
        let entitlementManager = EntitlementManager()
        let purchaseManager = PurchaseManager(entitlementManager: entitlementManager)

        self._entitlementManager = StateObject(wrappedValue: entitlementManager)
        self._purchaseManager = StateObject(wrappedValue: purchaseManager)
    }

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(entitlementManager)
                .environmentObject(purchaseManager)
                .task {
                    await purchaseManager.updatePurchasedProducts()
                }
        }
    }
}
```

```
struct ContentView: View {
    @EnvironmentObject
    private var entitlementManager: EntitlementManager

    @EnvironmentObject
    private var purchaseManager: PurchaseManager

    var body: some View {
```

```
9           VStack(spacing: 20) {
10              if entitlementManager.hasPro {
11                  Text("Thank you for purchasing pro!")
12              } else {
13                  Text("Products")
14                  ForEach(purchaseManager.products) { product in
```

The `ContentView` is reworked with the new `EntitlementManager` class. It has the same functionality as the previous step, but the feature unlock behavior is now fully moved outside `PurchaseManager`. `PurchaseManager` has one job, which is to deal with purchases and transactions; that means that extensions don't even need to know that it exists.

For extensions to check unlock status, `EntitlementManager` simply needs to be shared to the extension targets, which gives the ability for the extension to call `entitlementManager.hasPro`. See the snippet below for the exact code to use in the extensions:

```
1  let entitlementManager = EntitlementManager()
2  if entitlementManager.hasPro {
3      // Do something
4  } else {
5      // Don't do something
6  }
7
```

### Watch App

As mentioned above, companion watch apps behave differently than extensions because they do not benefit from receiving the same data from the iOS app with `Transaction.currentEntitlements`. Watch apps are also not able to use `UserDefaults` that are shared via an app group, which means that `EntitlementManager` cannot be used directly on a watchOS app. However, the watch app can use `WatchConnectivity` to communicate with the main iOS app, which can return the entitlement statuses from `EntitlementManager`. This may not be a feature that most developers need to use, but it's good to know how a watchOS app can differ from other iOS app extensions.

The full implementation up to this step can be seen here.

## Step 8: Handling renewals, cancellations, billing issues, and more

This is another area where `Transaction.currentEntitlements` shines. As mentioned earlier, `Transaction.currentEntitlements` will return the latest transaction for each active auto-renewable subscription and the latest transaction for each non-renewing subscription.

This means that whenever the app is opened, `Transaction.currentEntitlements` will have updated transactions that reflect any renewals, cancellations, or billing issues. Users will keep their entitlements when subscriptions renew and lose them if they cancel. Users will also lose their subscriptions if there are any unresolved billing issues.

Even though the app will make sure that the user is receiving the correct access for the current status of their subscriptions, this may not result in the best user experience. Renewals and cancellations are events that users don't need to be notified about, but a user will want to be notified of a billing issue. Billing issues are usually unplanned events that users will want to take care of so there is no lapse in the subscription.

On-device subscription handling cannot nicely inform a user of billing issues and grace periods. The subscription status may always be up to date, but this is a case where handling subscription server-side would be beneficial. Server-side subscription handling can more quickly detect billing issues and grace periods and allow apps to inform users.

## Step 9: Validating receipts

Validating receipts has historically been a *really* big deal when integrating StoreKit. With StoreKit 1, validating and parsing the receipt was the only way to determine purchases and what to unlock for users. Apple has a whole documentation page for choosing the best way to validate receipts, which mentions that there are two ways to verify a receipt's authenticity:

> Local, on-device receipt validation, which is best for validating the signature of the receipt for apps with in-app purchases
>
> Server-side receipt validation with the App Store, which works best for persisting in-app purchases to maintain and manage purchase records

Up to this point, this tutorial has been focused on a full local implementation of StoreKit without any server-side components. This leaves local on-device validation as the only way to verify a receipt's authenticity. There are also plenty of existing blog posts on how to validate and parse a receipt but, thankfully, any thought of the receipt can be ignored because of improvements made with StoreKit 2.

The reason why a receipt has not been mentioned much up to this point is because StoreKit 2 encapsulates all parsing and validating inside of `Transaction.currentEntitlements` and `Transaction.all` so developers don't need to worry about any of it.

## Step 10: Supporting in-app purchases from the App Store

In-app purchases aren't always initiated from within the app: There are flows where the user can purchase in-app products directly from the App Store. Developers can set promoted in-app purchases, which are displayed on the app's product page. When a user taps on the in-app products from the App Store page, the app opens up and should prompt the user to finalize the purchase. However, iOS does not automatically continue the user's action to purchase the product. The developer needs to add a listener to detect when this type of behavior happens, but adding this listener cannot be done with StoreKit 2 as of iOS 16.

The ability to continue the transaction from the App Store can only be done with StoreKit 1 APIs:

1. Set up a class `SKPaymentTransactionObserver` protocol (must inherit from `NSObject`).
2. Implement a `paymentQueue(_:shouldAddStorePayment:for:)` function.
3. Add `SKPaymentTransactionObserver` to the `SKPaymentQueue`.

```
@MainActor
class PurchaseManager: NSObject, ObservableObject {

    ...

    init(entitlementManager: EntitlementManager) {
        self.entitlementManager = entitlementManager
        super.init()
        SKPaymentQueue.default().add(self)
    }

    ...
}

extension PurchaseManager: SKPaymentTransactionObserver {
    func paymentQueue(_ queue: SKPaymentQueue, updatedTransactions transactions: [SKPay

    }
```

```
19
20      func paymentQueue(_ queue: SKPaymentQueue, shouldAddStorePayment payment: SKPayment
21          return true
22      }
23  }
24
```

This implementation (where `paymentQueue(_:shouldAddStorePayment:for:)` returns true) will continue the transaction as soon as the user enters the app. This is the easiest way to make this flow work, but there are times when instantly continuing the transaction might not be best behavior. If this is the case, the transactions can be deferred to a later (and better) time by returning false and manually adding the `SKPayment` to the `SKPaymentQueue` whenever it best suits the app.

The full implementation of this step can be seen here.

## Wrapping up this on-device subscription tutorial

With that, the tutorial for implementing in-app purchases purely on-device is complete. A user can see a list of products to choose from, purchase a product, unlock in-app content or features, restore purchases, use the unlocks in app extensions, and also purchase promoted products from the App Store. All this was implemented in the app using Swift with mostly StoreKit 2, but some StoreKit 1.

This type of StoreKit implementation can work really well for some types of apps. For example, it can work well if the app is only on Apple platforms and uses Universal Purchases. Users will be able to purchase IAPs and get the expected content or feature unlocks in the app.

A pure on-device StoreKit implementation may not be ideal of some types of apps, though. Subscription status cannot be shared across platforms to web or other native platforms. This method does not easily provide any insight or user behavior to the developers. Users cannot be directly notified outside the app (via email or other means) of billing errors or win-back campaigns on cancellation. A lot of data and hidden opportunities are caught inside all the magic of StoreKit 2.

The next section of this tutorial will cover a little bit of what it takes to implement StoreKit 2 with a custom server backend. Continue reading to find out if that is the best solution for you.

# Implementing StoreKit 2 with a server

In the steps above, we implemented a full subscription app without needing to write any code outside of the app, but that's not the only way to go. Besides the StoreKit 2 native APIs, Apple offers App Store Server APIs to get transaction history and subscription status.

This part of the tutorial will discuss how to design a web server to communicate with the App Store Server and how to pair an iOS app with the web server.

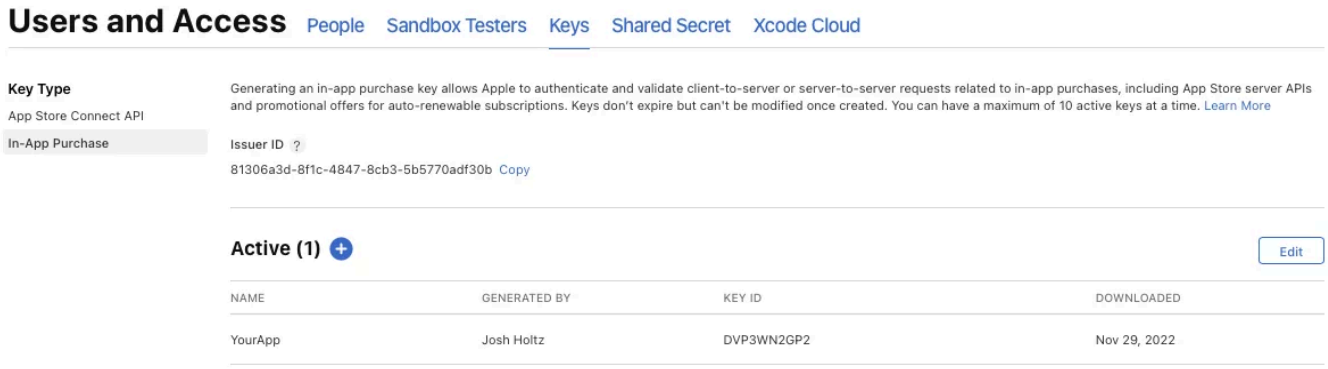
The sample backend will be written in Vapor (a Swift web server) with stubbed out functions and routes. The custom backend can be found here.

## App Store Server

This is the appropriately named service for communicating with the App Store about in-app purchases and subscriptions. It can be interacted with in two ways: by calling the App Store Server API and by consuming App Store Server Notifications.

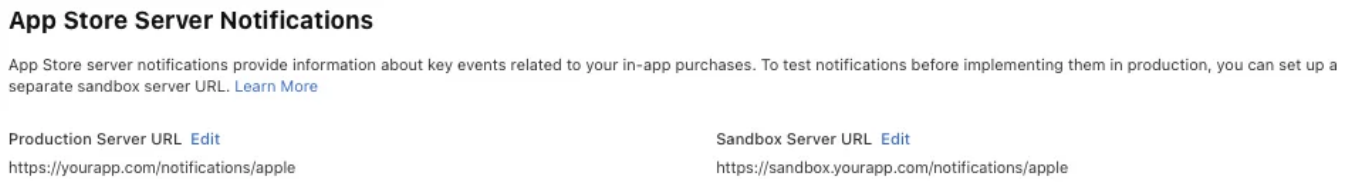
### App Store Server API

The App Store Server API is a REST API to request information about customers' in-app purchases. This API can be used to get a customer's transaction history, see the status of all auto-renewable subscriptions, update consumable IAP status, look up orders, get refund history, and extend a subscription renewal date. Most of these endpoints require an original transaction identifier, which is obtained from the `Transaction` object after making a purchase from the native StoreKit 2 API.
Requests to the App Store Server API are authorized by generating a JSON Web Token (JWT). Secret keys are required to generate a JWT and can be created through App Store Connect.




### App Store Server Notifications

App Store Server Notifications provide a way to monitor real-time in-app purchase events with direct notifications from the App Store. The web server's HTTPS URL needs to be set through App Store Connect for App Store Server Notifications to know where to send the request. A different URL can be set for each of the production and sandbox environments.




There are 15 different types of notifications that will be sent from version 2 of App Store Server Notifications:

CONSUMPTION_REQUEST

DID_CHANGE_RENEWAL_PREF

DID_CHANGE_RENEWAL_STATUS

DID_FAIL_TO_RENEW

DID_RENEW

EXPIRED

GRACE_PERIOD_EXPIRED

OFFER_REDEEMED

PRICE_INCREASE

REFUND

REFUND_DECLINED

RENEWAL_EXTENDED

REVOKE

SUBSCRIBED

TEST

Requests from App Store Server Notifications are sent as an HTTP POST, where the payload is signed by the App Store in a JSON Web Signature (JWS) format. JWS is used to increase security and trust that the payload was sent by the App Store.

The notification payload contains:

- Notification type
- Notification subtype
- Notification UUID
- Data:
  - App Apple ID
  - Bundle ID
  - Bundler version
  - Environment
  - Renewal Info
  - Transaction Info

To test the connection between App Store and the backend listening for App Store Server Notifications, a test event can be requested from the App Store Server API. This is the easiest way to do an end-to-end connection test. Before this method existed, a test had to be done by purchasing a product from either a sandbox or production app.

## Designing the system

With App Store Server API and App Store Server Notifications in mind, it is now time to design the system.

### Purchasing products

Product purchasing still needs to be done with the `Products.purchase(product)` through the native StoreKit 2 API, but things diverge quickly after that. The app will no longer rely on StoreKit 2's `Transaction.currentEntitlements` for determining which products are currently active; instead, it will post the transaction information to the backend. The backend requires the original transaction identifier to be able to fetch transaction history from the App Store Server API and match updated transactions from App Store Server Notifications for the user. This provides the advantage of being able to independently verify the purchase as well as giving the backend the concept of subscriptions..

Along with posting the transaction after a successful purchase, the app will also need to post transaction updates it discovers while listening to `Transaction.updates`. This is needed for the `.pending` purchases that were waiting for Strong Customer Authentication or Ask to Buy that didn't have transaction identifiers at the time of purchase.

Below is an example of what the changes to the iOS app might look like.

```swift
func purchase(_ product: Product) async throws {
    let result = try await product.purchase()

    switch result {
    case let .success(.verified(transaction)):
        await transaction.finish()
        await postTransaction(transaction)
    case let .success(.unverified(_, error)):
        break
    case .pending:
        break
    case .userCancelled:
        break
    @unknown default:
        break
    }
}

private func observeTransactionUpdates() -> Task<Void, Never> {
    Task(priority: .background) {
        for await result in Transaction.updates {
            guard case .verified(let transaction) = result else {
                continue
            }

            guard let product = products.first(where: { product in
                product.id == transaction.productID
            }) else { continue }
            await postTransaction(transaction, product: product)
        }
    }
}

func postTransaction(_ transaction: Transaction, product: Product) async {
    let originalTransactionID = transaction.originalID
    let body: [String: Any] = [
        "original_transaction_id": originalTransactionID,
        "price": product.price,
        "display_price": product.displayPrice
    ]
    await httpPost(body: body)
```

```
42  }
43
```

The sample Vapor backend will get a new endpoint to post the transaction to. The endpoint should end up doing all of the following:

- Ensuring that the app user is logged in
- Sending a request to App Store Server to validate and fetch transaction information
- Storing important transaction information in the database
- Determining and updating unlocked entitlements or features for the logged-in user

Below is the start of what the sample Vapor backend will look like.

```
1   app.post("apple/transactions") { req async -> Response in
2       do {
3           let newTransaction = try req.content.decode(PostTransaction.self)
4           let transaction = processTransaction(
5               originalTransactionID: newTransaction.originalTransactionId)
6           return try await transaction.encodeResponse(for: req)
7       } catch {
8           return .init(status: .badRequest)
9       }
10  }
11
```

Note that all transaction updates (renewals, cancellations, etc) will also be observed through this function. This means that the backend could be getting duplicate transactions from the app and in real time from App Store Server Notifications. The backend will need to be able to handle this behavior.

### Handling product prices

Storing the price with a transaction is important to understanding lifetime value (LTV) of a customer and making decisions on user acquisition spending. The best way to do this is to maintain a total of the user's spending in the transaction history.

It may be surprising to learn that App Store Server doesn't include the price or currency of the product in transactions. This means that the iOS app needs to send up the product price. A StoreKit 2 `Product` object has `price` (numeric value) and `displayPrice` (string value)

properties that can be used for this purpose. StoreKit 2 (as of iOS 16) doesn't have a way to directly get the currency used to purchase the product. Sending up `displayPrice` gives the backend an opportunity to manually parse the currency based on the symbol in the string. If manually parsing the currency isn't good enough, it is possible to use StoreKit 1 to get the currency from the `SKProduct` object. It's messy and more work, but it will be more reliable.

### Handling renewals, cancellations, billing issues, and more

The app is now dependent on the custom backend to keep all subscriptions and transactions up to date. The backend should be made aware of renewals, cancellations, billing issues, and refunds as soon as possible to ensure the app user is not getting features or services that they aren't paying for. There are two ways to do this:

- Listening for real-time notifications from App Store Server Notifications
- Polling App Store Server API

While it's not necessary to implement both, doing so will make for a much stronger, more robust, and more trustworthy source of truth.

#### Listening to App Store Server Notifications

Enabling App Store Server Notifications is the best practice and the most efficient way of keeping subscription status and refunded transactions up to date. Apple sends a HTTP POST to a URL the developer has defined in App Store Connect with information about an updated transaction as soon as it's been updated. This will ensure that any system listening to App Store Server Notifications matches the transaction information that Apple has on file.

The code snippet below shows how to start handling notifications from App Store Server:

1. The request is posted to `https://<mydomain>.com/apple/notifications`.
2. The request payload is decoded into a `SignedPayload` Swift struct that represents responseBodyV2. The struct contains a signed payload (JWS).
3. The signed payload is decoded using Apple's root certificate into a `NotificationPayload` Swift struct that represents responseBodyV2DecodedPayload.
4. The `NotificationPayload` is then sent to a worker queue to process the transaction away from the web server's main process. This is where transaction and entitlement database updates for user access would be executed and also where emails or other notifications could be sent to use's to let them know of any billing issues, expirations, or possible win-back offers.

```
struct SignedPayload: Decodable {
    let signedPayload: String
}

struct NotificationPayload: JWTPayload {
    let notificationType: NotificationType
    let notificatonSubtype: NotificationSubtype?
    let notificationUUID: String
    let data: NotificationData
    let version: String
    let signedDate: Int

    func verify(using signer: JWTSigner) throws {
    }
}

app.post("apple/notifications") { req async -> HTTPStatus in
    do {
        let notification = try req.content.decode(SignedPayload.self)

        let payload = try req.application.jwt.signers.verifyJWSWithX5C(
            notification.signedPayload,
            as: NotificationPayload.self,
            rootCert: appleRootCert)

        // Add job to process and update transaction
        // Updates user's entitlements (unlocked products or features)
        try await req.queue.dispatch(AppleNotificationJob.self, payload)

        return .ok
    } catch {
        return .badRequest
    }
}

```

The full model for `SignedPayload` can be found here.

One of the riskiest parts of relying on App Store Server Notifications is missing out on notifications. If App Store Server doesn't receive a response or detects a bad HTTP response from the web request, it will retry sending the notification up to five more times. These retries will occur at 1, 12, 24, 48, and 72 hours after the previous attempt. The most likely case of this

would happen because of server or service outages. Apple has another solution for this with fetching notification history through the App Store Server API. If any of those retries are missed, the App Store Server API can be used to replay through these notifications to get caught back up.

There is more extensive documentation on how to enable, receive, and respond to notifications that are out of the scope of this tutorial.

### Polling App Store Server API

Even while consuming real-time notifications, occasionally polling the App Store Server API for transaction updates is essential for a robust system. The processing of the transaction is very similar to real-time notifications, but the tricky part with this approach is determining how often to schedule these API calls to App Store Server.

The most obvious approach would be scheduling auto-renewable subscriptions just after their time of expiration or renewal. Subscription statuses should also be checked before their renewal to see if they've been canceled, which allows warning the user about service ending soon or enables attempting to win back the user with some deal. There is no perfect moment to do this since a user can cancel at any time. This polling could start off infrequently and get more frequent as the expiration or renewal time approaches.

Polling will also need to be rescheduled after a renewal period if a billing issue occurs due to there being a grace period. This allows the user to keep service a few days longer if they are able to fix their billing information.

Non-renewable subscriptions and non-consumables should also be polled for updates. Even though these types of events don't have cancellation or renewal events, a user could ask for a refund which could revoke access to service or content.

Polling seems technically easy at first but can quickly become a complex scheduling dance, which is why using App Store Server Notifications is the best practice: Apple will only notify about things that change and the exact time they change.

## Listing products

Available products for purchase will be returned by the API. These product identifiers would ideally be stored in a database to be easily configurable whenever the app's paywall needs to change. The product identifiers being served from the API would even allow for A/B testing of paywalls with users to determine which list of products work best. Any solution here is better

than hardcoding a product identifier into the app that can only be changed through an app update.

### Benefits and tradeoffs

The native StoreKit 2 API hides a lot of the intricacies of in-app purchases and subscriptions when it comes to determining current entitlements. There are valid reasons to only use the native API, but there are many more to use a backend to manage subscriptions, including these:

- Receive real-time notifications of subscription status changes and refunds
- Validate transactions and receipts
- Manage customers and offer refunds and subscription extensions
- Analyze revenue and churn statistics outside App Store Connect
- Bind subscription status to backend service features
- Share purchases across the web and other mobile platforms

Developing a server-side implementation for handling IAPs with StoreKit 2 gets exponentially more complex compared to a pure on-device implementation. Because of that, this part of the tutorial isn't going to be a complete step-by-step guide to building a custom subscription backend.

## Final notes

Our biggest goal at RevenueCat is to help developers make more money. We do this by offering an entire **mobile subscription platform** with native and **cross-platform SDKs** to make integrating in-app purchases as easy as possible in any application. Our customers don't need to worry about the different native subscription APIs or the intricacies of keeping subscription status in sync with renewals, cancellations, and billing issues.

We built this StoreKit 2 tutorial because, even though we'd love to have everyone as a customer, we know that sometimes it's better to build your own. At the end of the day, we want all developers to succeed.

For Marketing Teams

For Product Teams

Pricing

verifyReceipt tool

Fair Billing Policy