Sample Code

Offering, completing, and restoring in-app purchases

Fetch, display, purchase, validate, and finish transactions in your app.

Download

iOS 16.0+ | iPadOS 16.0+ | macOS 13.0+ | tvOS 16.0+ | Xcode 15.0+

Overview

Use the In-App Purchase API to give people the ability to purchase virtual goods within your app or directly from the App Store using the StoreKit framework. This sample code project demonstrates how to retrieve, display, and restore in-app purchases. First, you set up your app to register and use a single-transaction queue observer at launch. The transaction queue observer manages all payment transactions and handles all transaction states. Confirm that it's a shared instance of a custom class that conforms to the `SKPaymentTransactionObserver` protocol. Then, remove the transaction observer when the system is about to terminate the app. See `Setting Up the Transaction Observer and Payment Queue` for more information.

This sample code project, which builds the InAppPurchases app, supports the iOS, iPadOS, macOS, and tvOS platforms. After launching, the app queries the App Store about product identifiers in the `Products.plist` file. The app updates its UI with the App Store's response, which may include available products for sale, unrecognized product identifiers, or both. The app also displays all available purchased and restored payment transactions.

Configure the sample code project

Before you can run and test this sample code project, you need to:

1. Start with a completed app that supports in-app purchases and has some configured in-app purchases in App Store Connect. For more information, see Overview for configuring in-app

purchases.

2. Create a Sandbox Apple Account in App Store Connect.
3. Open the sample code project in Xcode, select the target that you want to build, click Signing & Capabilities, and change its bundle identifier to one that supports in-app purchases.
4. Choose the appropriate team from the Team pop-up menu to let Xcode automatically manage your provisioning profile. See Assign a project to a team for details.
5. Open the `ProductIds.plist` file in the sample and update its content with your existing in-app purchases product IDs.
6. For iOS and tvOS devices, build and run the `InAppPurchases` and `InAppPurchasestvOS` targets, respectively, which the sample uses to build the app. If you have any code-signing issues, see If a code signing error occurs.
7. For macOS, before building the `InAppPurchasesmacOS` target, sign out of the Mac App Store. Build the target, then launch the resulting app from the Finder the first time to obtain a receipt. See `Sign in to the App Store with your Sandbox Apple Account` for details.
8. Upon launching, the app queries the App Store about the product identifiers in `ProductIds.plist`. When successful, it displays a list of products available for sale in the App Store. Tap any product in that list to purchase it. When you receive a prompt to authenticate the purchase, use your test user account. If the product request fails, see `invalidProductIdentifiers` for reasons the App Store may return invalid product identifiers.

Display available products for sale with localized pricing

The sample configures the app so it confirms that the user has authorization to make payments on the device before presenting products for sale.

```
var isAuthorizedForPayments: Bool {
    return SKPaymentQueue.canMakePayments()
}
```

After the app confirms authorization, it sends a products request to the App Store to fetch localized product information. Querying the App Store ensures that the app only presents users with products available for purchase. The app initializes the products request with a list of product identifiers associated with products to sell in its UI. Be sure to keep a strong reference to the products request object; the system may release it before the request completes. See Product ID for more information.

```
fileprivate func fetchProducts(matchingIdentifiers identifiers: [String]) {
    // Create a set for the product identifiers.
    let productIdentifiers = Set(identifiers)

    // Initialize the product request with the above identifiers.
    productRequest = SKProductsRequest(productIdentifiers: productIdentifiers)
    productRequest.delegate = self

    // Send the request to the App Store.
    productRequest.start()
}
```

The App Store responds to the products request with an `SKProductsResponse` object. Its `products` property contains information about all the products that are available for purchase in the App Store. The app uses this property to update its UI. The response's `invalidProductIdentifiers` property includes all product identifiers that the App Store doesn't recognize. See `invalidProductIdentifiers` for reasons the App Store may return invalid product identifiers.

```
// Contains products with identifiers that the App Store recognizes. As such, they a
if !response.products.isEmpty {
    availableProducts = response.products
}

// invalidProductIdentifiers contains all product identifiers that the App Store doe
if !response.invalidProductIdentifiers.isEmpty {
    invalidProductIdentifiers = response.invalidProductIdentifiers
}
```

To display the price of a product in the UI, the app uses the locale and currency that the App Store returns. For instance, consider a user who is logged in to the French App Store and their device uses the United States locale. When attempting to purchase a product, the App Store displays the product's price in Euros. Converting and showing the product's price in U.S. dollars to match the device's locale would be incorrect.

```
extension SKProduct {
    /// - returns: The cost of the product formatted in the local currency.
    var regularPrice: String? {
        let formatter = NumberFormatter()
        formatter.numberStyle = .currency
        formatter.locale = self.priceLocale
        return formatter.string(from: self.price)
```

```
    }
}
```

Interact with the app to purchase products

Users can tap any product available for sale in the UI to purchase it. The app allows users to restore non-consumable products and auto-renewable subscriptions. The sample implements the Restore button and Settings > Restore all restorable purchases to implement this feature in the iOS and tvOS version of the app, respectively. It implements the Store > Restore menu item to restore purchases in the macOS version of the app.

Tapping any purchased item brings up purchase information, such as product identifier, transaction identifier, and transaction date. When the purchase is a restored one, the purchase information also contains its original transaction's identifier and date.

Handle payment transaction states

When a transaction is pending in the payment queue, StoreKit notifies the app's transaction observer by calling its `paymentQueue(_:updatedTransactions:)` method. Every transaction has five possible states, including `SKPaymentTransactionState.purchasing`, `SKPaymentTransactionState.purchased`, `SKPaymentTransactionState.failed`, `SKPaymentTransactionState.restored`, and `SKPaymentTransactionState.deferred`. For more information, see `SKPaymentTransactionState`. The observer's `paymentQueue(_:updatedTransactions:)` needs to be able to respond to any of these states at any time.

```
func paymentQueue(_ queue: SKPaymentQueue, updatedTransactions transactions: [SKPaym
    for transaction in transactions {
        switch transaction.transactionState {
        case .purchasing: break
        // Don't block the UI. Allow the user to continue using the app.
        case .deferred: print(Messages.deferred)
        // The purchase was successful.
        case .purchased: handlePurchased(transaction)
        // The transaction failed.
        case .failed: handleFailed(transaction)
        // There are restored products.
        case .restored: handleRestored(transaction)
        @unknown default: fatalError(Messages.unknownPaymentTransaction)
        }
    }
}
```

When a transaction fails, the app inspects the `error` property to determine what happened. The app only displays errors with code that is different from `paymentCancelled`.

```
// Don't send any notifications when the user cancels the purchase.
if (transaction.error as? SKError)?.code != .paymentCancelled {
    DispatchQueue.main.async {
        self.delegate?.storeObserverDidReceiveMessage(message)
    }
}
```

When the user defers a transaction, apps need to allow them to continue using the UI while waiting for StoreKit to update the transaction.

Restore completed purchases

When users purchase non-consumables, auto-renewable subscriptions, or non-renewing subscriptions, they expect them to be available on all their devices indefinitely. The app provides a UI that allows users to restore their past purchases. See `Understand Product Types` for more information.

```
@IBAction func restore(_ sender: UIBarButtonItem) {
    // Calls StoreObserver to restore all restorable purchases.
    StoreObserver.shared.restore()
```

```
}
```

The sample uses `restoreCompletedTransactions()` to restore non-consumables and auto-renewable subscriptions. StoreKit notifies the app's transaction observer by calling `paymentQueue(_:updatedTransactions:)` with a transaction state of `.restored` for each restored transaction. Restoring non-renewing subscriptions isn't within the scope of this sample code project. For information about restore failures, see `restoreCompletedTransactions()`.

Provide content and finish the transaction

Apps need to deliver the content or unlock the purchased functionality after receiving a transaction with a state of `.purchased` or `.restored`. These states indicate that the App Store has received a payment for a product from the user.

Unfinished transactions stay in the payment queue. StoreKit calls the app's persistent observer's `paymentQueue(_:updatedTransactions:)` each time upon launching or resuming from the background until the app finishes these transactions. As a result, the App Store may repeatedly prompt users to authenticate their purchases or prevent them from purchasing products from the app.

The sample calls `finishTransaction(_:)` on transactions with a state of `.failed`, `.purchased`, or `.restored` to remove them from the queue. Finished transactions aren't recoverable. Therefore, apps need to provide the purchased content or complete their purchase process before finishing transactions.

```
// Finish the successful transaction.
SKPaymentQueue.default().finishTransaction(transaction)
```

See Also

Essentials

Setting up the transaction observer for the payment queue

Enable your app to receive and handle transactions by adding an observer.

~~class SKPaymentQueue~~

A queue of payment transactions for the App Store to process.

Deprecated

~~protocol SKPaymentTransactionObserver~~

A set of methods that process transactions, unlock purchased functionality, and continue promoted In-App Purchases.

Deprecated

~~`protocol` SKPaymentQueueDelegate~~

The protocol that provides information needed to complete transactions.

Deprecated

~~`class` SKRequest~~

An abstract class that represents a request to the App Store.

Deprecated