In-App Purchase Programming Guide

Developer

Contents

Figures and Tables

Introduction

In-App Purchase allows you to embed a store directly within your application. You implement In-App Purchase in your application using the Store Kit framework. Store Kit connects to the App Store on your application's behalf to securely process payments from the user. Store Kit prompts the user to authorize the payment, then notifies your application so that it can provide items the user purchased. You can use this in-application payment functionality to collect payment for enhanced functionality or additional content usable by your application.

For example, you could use In-App Purchase to implement any of the following scenarios:

- A basic version of your application with additional premium features.
- A book reader application that allows the user to purchase and download new books.
- A game that offers new environments (levels) to explore.
- An online game that allows the player to purchase virtual property.

Important: In-App Purchase only collects payment. You must provide any additional functionality, including unlocking built-in features or downloading content from your own servers. This documentation details the technical requirements of adding a store to your application. For more information on the business requirements of using In-App Purchase, see the App Store Resource Center. You must also read your licensing agreement for the definitive treatment of what you may sell and how you are required to provide those products in your application.

Who Should Read This Document

You should read this if you are interested in offering additional paid functionality to users from within your application.

Organization of This Document

This document contains the following chapters:

- "Overview of In-App Purchase" (page 7) introduces the functionality offered by In-App Purchase.

- "Retrieving Product Information" (page 14) describes how your application retrieves information from the App Store about products it offers.
- "Making a Purchase" (page 17) explains how your application requests payment from the App Store.
- "Adding a Store to Your Application" (page 20) is a walkthrough that describes how to add a store to your application.
- "Verifying Store Receipts" (page 25) describes how your server can verify that a receipt came from the App Store.
- "Testing a Store" (page 28) discusses how to use the sandbox environment to test your application.
- "Auto-Renewable Subscriptions" (page 30) describes how your application can implement subscriptions using the Apple Store to manage the renewal process for you.

See Also

The App Store Resource Center describes the business side of using In-App Purchase, as well as the steps you need to take to sell a product within your application.

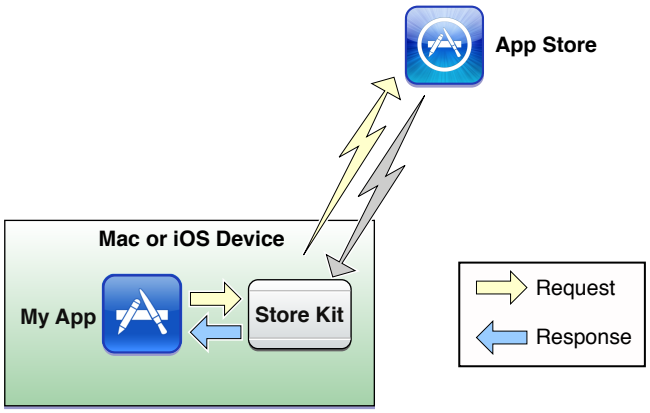
The *iTunes Connect Developer Guide* describes how to configure products and test accounts on the App Store.

The *Store Kit Framework Reference* describes the API for interacting with the App Store.

Overview of In-App Purchase

Store Kit communicates with the App Store on behalf of your application. Your application uses Store Kit to receive localized information from the App Store about products you want to offer in your application. Your application displays this information to users and allows them to purchase items. When a user wants to purchase an item, your app calls Store Kit to collect payment from the user. Figure 1-1 shows the basic store model.

Figure 1-1 In-App Store model



The Store Kit API is only a small part of the process of adding a store to your application. You need to decide how to track the products you plan to deliver, how your application presents a store front to the user, and how your application delivers the products users purchase from your store. The rest of this chapter explains the process of creating products and adding a store to your application.

Products

A **product** is any feature that you want to sell in your application's store. Products are associated with the App Store through iTunes Connect in the same way that you create new applications. There are four supported kinds of products that you may sell using In-App Purchase:

- *Content* includes digital books, magazines, photos, artwork, game levels, game characters, and other digital content that can be delivered within your application.
- *Functionality* products unlock or expand features you've already delivered in your application. For example, you could ship a game with multiple smaller games that could be purchased by the user.

- *Services* allow your application to charge users for one-time services, such as voice transcription. Each time the service is used is a separate purchase.
- *Subscriptions* provide access to content or services on an extended basis. For example, your application might offer monthly access to financial information or to an online game portal.

In-App Purchase provides a general mechanism for creating products, leaving the specifics of how your products are implemented up to you. However, there are few important guidelines to keep in mind as you design your application:

- You must deliver a digital good or service within your application. Do not use In-App Purchase to sell real-world goods and services.
- You may not offer items that represent intermediary currency because it is important that users know the specific good or service they are buying.
- Items you offer for purchase may not contain, or relate to, pornography, hate speech, defamation, or gambling (simulated gambling is acceptable).

For detailed information about what can be offered using In-App Purchase, consult your licensing agreement.

Registering Products with the App Store

Every product you wish to offer in your store must first be registered with the App Store through iTunes Connect. When you register a product, you provide a name, description, and pricing for your product, as well as other metadata used by the App Store and your application.

You identify a particular product using a unique string called a **product identifier**. When your application uses Store Kit to communicate with the App Store, it uses product identifiers to retrieve the configuration data you provided for the product. Later, when a customer wants to purchase a product, your application identifies the product to be purchased using its product identifier.

The App Store supports many types of products:

- **Consumable** products must be purchased each time the user needs that item. For example, one-time services are commonly implemented as consumable products.
- **Non-consumable** products are purchased only once by a particular user. Once a non-consumable product is purchased, it is provided to all devices associated with that user's iTunes account. Store Kit provides built-in support to restore non-consumable products on multiple devices.
- **Auto-renewable subscriptions** are delivered to all of a user's devices in the same way as non-consumable products. However, auto-renewable subscriptions differ in other ways. When you create an auto-renewable subscription in iTunes Connect, you choose the duration of the subscription. The App Store automatically

renews the subscription each time its term expires. If the user chooses to not allow the subscription to be renewed, the user's access to the subscription is revoked after the subscription expires. Your application is responsible for validating whether a subscription is currently active and can also receive an updated receipt for the most recent transaction.

- **Free subscriptions** are a way for you to put free subscription content in Newsstand. Once a user signs up for a free subscription, the content is available on all devices associated with the user's Apple ID. Free subscriptions do not expire and can only be offered in Newsstand-enabled apps.
- **Non-renewing subscriptions** are a mechanism for creating products with a limited duration. Non-renewing subscriptions differ from auto-renewable subscriptions in a few key ways:
    - The term of the subscription is not declared when you create the product in iTunes Connect; your application is responsible for providing this information to the user. In most cases, you would include the term of the subscription in the description of your product.
    - Non-renewing subscriptions may be purchased multiple times (like a consumable product) and are not automatically renewed by the App Store. You are responsible for implementing the renewal process inside your application. Specifically, your application must recognize when the subscription has expired and prompt the user to purchase the product again.
    - You are required to deliver non-renewing subscriptions to all devices owned by the user. Non-renewing subscriptions are not automatically synchronized to all devices by Store Kit; you must implement this infrastructure yourself. For example, most subscriptions are provided by an external server; your server would need to implement a mechanism to identify users and associate subscription purchases with the user who purchased them.

Detailed information about registering products with the App Store can be found in *iTunes Connect Developer Guide* .

Feature Delivery

The delivery mechanism your application uses to provide products to users has significant implications on its design and implementation. There are two basic models you should expect to use to deliver products to users: the built-in model and the server model. In both models, you track the list of products offered in the store and deliver products successfully purchased by users.

Built-in Product Model

In the built-in product model, everything required to deliver products is built in to your application. This model is most often used to unlock functionality in your application. You could also use this model to deliver content provided in your application's bundle. A key advantage of this model is that your application can promptly deliver products to the customer. Most built-in products should be non-consumable.

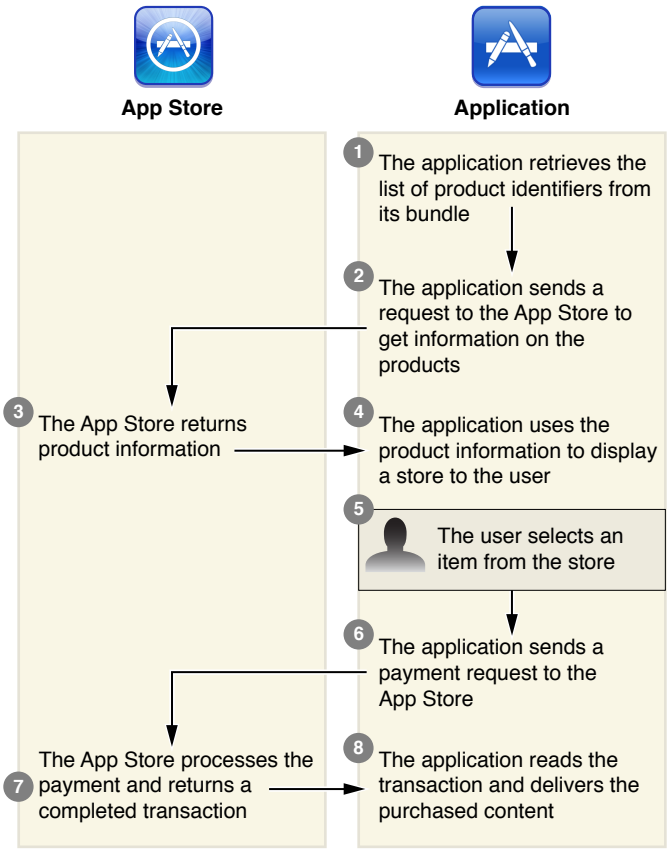
Important: In-App Purchase does not provide the capability for your application to be patched after a successful purchase. If your product requires changes to your application's bundle, you must deliver an updated version of your application to the App Store.

To identify products, your application stores the product identifiers in your application's bundle. Apple recommends using a property list (plist) to track product identifiers for your built-in features. Content-driven applications can use this to add new content without modifying the source for your application.

After a product is successfully purchased, your application must unlock the feature and deliver it to the user. The simplest way to unlock features is by changing your application preferences. See "Implementing Application Preferences". Application preferences are backed up when users backs up their iOS-based devices. Your application may want to recommend to users that they back up their devices after making a purchase to ensure that purchases are not lost.

Figure 1-2 shows the series of actions your application takes to deliver a built-in product.

Figure 1-2 Built-in product delivery

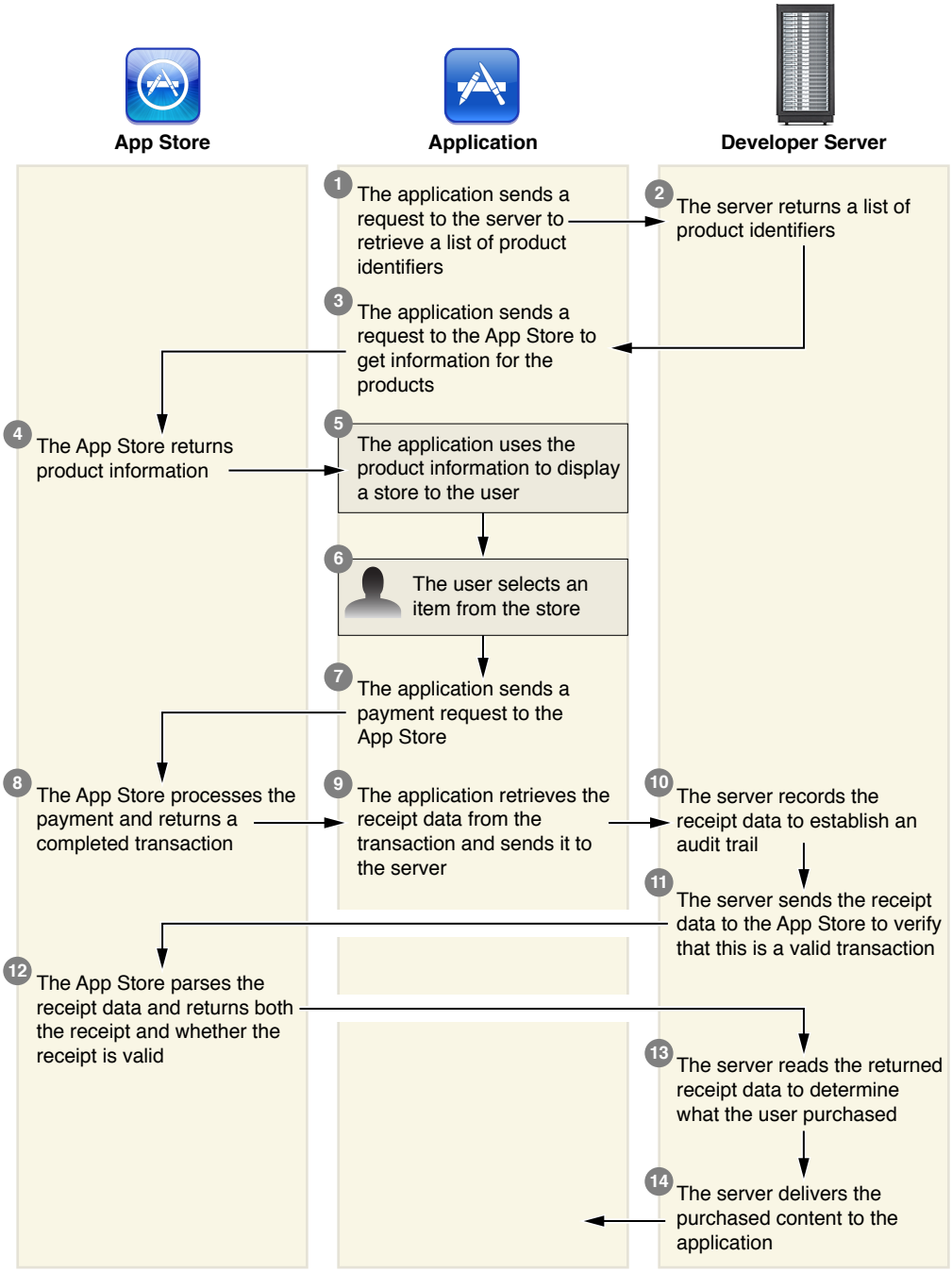


Server Product Model

In the server product model, you provide a separate server that delivers products to your application. Server delivery is appropriate for subscriptions, services and content, because these products can be delivered as data without altering your application bundle. For example, a game might deliver new play environments (puzzles or levels) to the application. Store Kit does not define the design of your server or its interactions with your application. *You are responsible for designing all interactions between your application and your server.* Further, Store Kit does not provide a mechanism to identify a particular user. Your design may require you to provide a mechanism to identify a user. If your application requires these (for example, to track which subscriptions are associated with a particular user), you need to design and implement this yourself.

Figure 1-3 expands the built-in model to show interactions with a server.

Figure 1-3 Server product delivery



Apple recommends you retrieve product identifiers from your server, rather than including them in a property list. This gives you the flexibility to add new products without updating your application.

In the server model, your application retrieves the signed receipt associated with a transaction and sends it to your server. Your server can then validate the receipt and decode it to determine which content to deliver to your application. This process is covered in detail in "Verifying Store Receipts" (page 25).

The server model has additional security and reliability concerns. You should test the entire environment for security threats. *Secure Coding Guide* provides additional recommendations.

Although non-consumable products may be recovered using the built-in capabilities of Store Kit, non-renewing subscriptions must be restored by your server. You are responsible for recording information about non-renewing subscriptions and restoring them to users. Optionally, consumable products could also be tracked by your server. For example, if your consumable product is a service provided by your server, you may want the user to retrieve the results of that request on multiple devices.

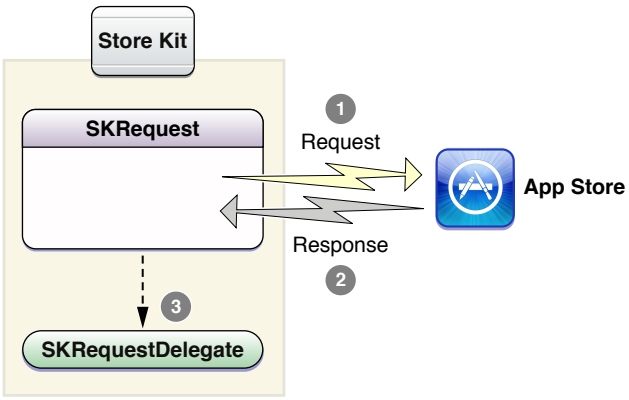
Retrieving Product Information

When your application is ready to display a store to the user, it must populate its user interface with information from the App Store. This chapter details how to request product details from the App Store.

Sending Requests to the App Store

Store Kit provides a common mechanism to request information from the App Store. Your application creates and initializes a request object, attaches a delegate to it, and starts the request. Starting a request transmits it to the App Store, where it is processed. When the App Store processes the request, the request's delegate is called asynchronously to deliver the results to your application. Figure 2-1 shows the request model.

Figure 2-1 Store Kit request model



If your application quits while a request is pending, your application needs to resend it.

SKRequest

`SKRequest` is an abstract base class for requests sent to the store.

SKRequestDelegate

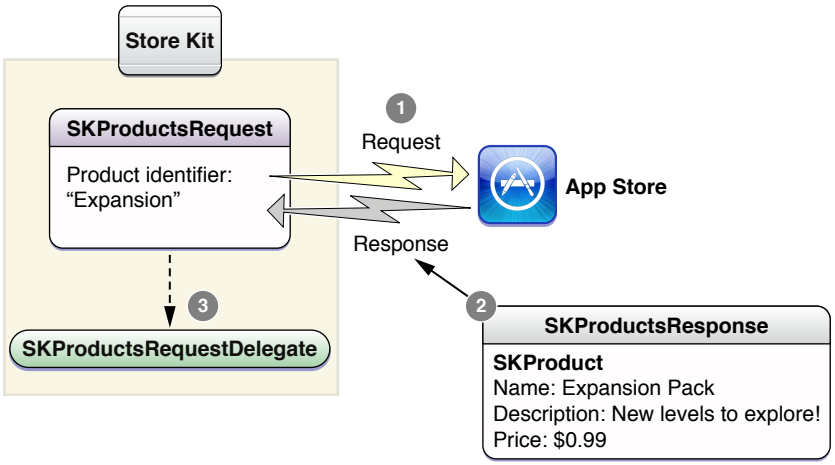
`SKRequestDelegate` is a protocol that your application implements to handle requests that completed successfully and requests that failed because of an error.

Requesting Information About Products

Your application uses a **products request** to retrieve localized information about a product. Your application creates a request that contains a list of product identifier strings. As described earlier, your application might embed product identifiers inside your application or it might retrieve the identifiers from an external server.

When you start the products request, the product identifier strings are transmitted to the App Store. The App Store responds with the localized information you previously entered in iTunes Connect. You use these details to populate the user interface of your store. Figure 2-2 shows a products request.

Figure 2-2 A request for localized product information



Important: You must make a product request for a particular product identifier before allowing the user to purchase that product. Retrieving product information from the App Store ensures that you are using a valid product identifier for a product you have marked available for sale in iTunes Connect.

SKProductsRequest

An `SKProductsRequest` object is created with a list of product identifier strings for the products you want to display in your store.

SKProductsRequestDelegate

The `SKProductsRequestDelegate` protocol is implemented by an object in your application to receive the response from the store. It receives the response asynchronously when the request is successfully processed.

SKProductsResponse

An `SKProductsResponse` object contains a `SKProduct` object for each valid product identifier in the original request as well as a list of the product identifiers that were not recognized by the store. The store might not recognize the identifier for a number of reasons; it might be misspelled, marked unavailable for sale, or changes you have made in iTunes Connect have not propagated to all of the App Store servers.

SKProduct

An `SKProduct` object provides localized information about a product you've registered with the App Store.

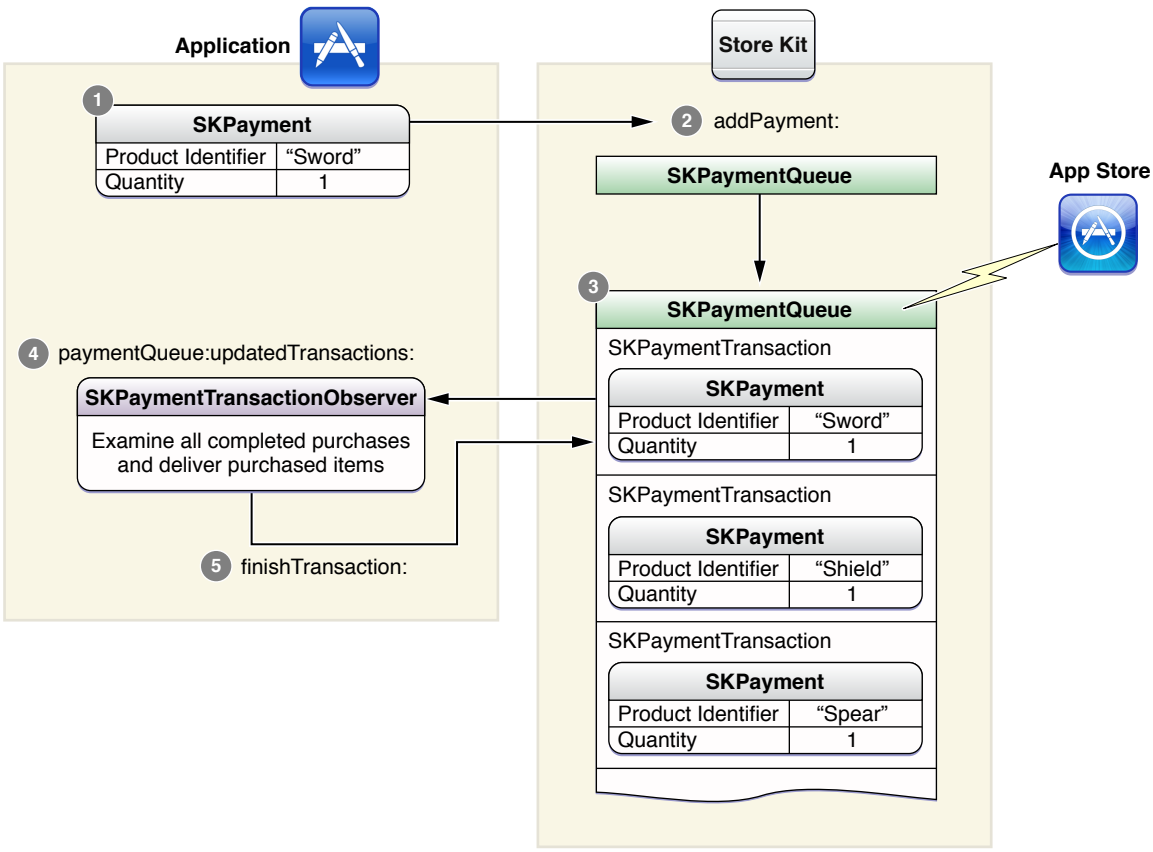
Making a Purchase

When the user is ready to purchase an item, your application asks the App Store to collect payment. When your application asks for payment, the App Store creates a persistent transaction and continues to process the payment, even if the user quits and relaunches your application. The App Store synchronizes the list of pending transactions with your application and delivers updates to your application when the status of any of these transactions changes.

Collecting Payments

To collect payment, your application creates a payment object and queues it on the payment queue, as shown in Figure 3-1.

Figure 3-1 Adding a payment request to the queue



When the payment is added to the payment queue, a persistent transaction is created to hold it. After the payment is processed, the transaction is updated with information about the payment collection. Your application implements an observer that receives messages when transactions are updated. The observer should provide purchased items to the user and then remove the transaction from the payment queue.

SKPayment

Collecting payment starts with a payment object. The payment object includes a product identifier and optionally includes the quantity of that product to be purchased. You can queue the same payment object more than once; each time a payment object is queued results in a separate request for payment.

Users can disable the ability to make purchases in the Settings application. Before attempting to queue a purchase, your application should first confirm that payment can be processed. You do this by calling the payment queue's `canMakePayments` method.

SKPaymentQueue

The payment queue is used to communicate with the App Store. When payments are added to the queue, Store Kit transmits the request to the App Store. Store Kit presents dialogs to ask the user to authorize payment. The completed transaction is returned to your application's observer.

SKPaymentTransaction

A transaction is created for every payment added to the queue. Each transaction has properties that allow your application to determine the status of the transaction. When payment is collected, the transaction includes additional details about the successful transaction.

Although your application can ask the payment queue for a list of pending transactions, it is more common for an application to wait until the payment queue notifies the payment queue's observer with a list of updated transactions.

SKPaymentTransactionObserver

Your application implements the `SKPaymentTransactionObserver` protocol on an object and adds it as an observer to the payment queue. The observer's primary responsibility is to examine completed transactions, deliver items that were successfully purchased, and remove those transactions from the payment queue.

Your application should associate an observer with the payment queue when it launches, rather than wait until the user attempts to purchase an item. Transactions are not lost when an application terminates. The next time the application launches, Store Kit resumes processing transactions. Adding the observer during your application's initialization ensures that all transactions are returned to your application.

Restoring Transactions

Once a transaction has been processed and removed from the queue, your application normally never sees it again. However, if your application supports product types that must be restorable, you must include an interface that allows users to restore these purchases. This interface allows a user to add the product to other devices or, if the original device was wiped, to restore the transaction on the original device.

Store Kit provides built-in functionality to restore transactions for non-consumable products, auto-renewable subscriptions and free subscriptions. To restore transactions, your application calls the payment queue's `restoreCompletedTransactions` method. The payment queue sends a request to the App Store to restore the transactions. In return, the App Store generates a new restore transaction for each transaction that was previously completed. The restore transaction object's `originalTransaction` property holds a copy of the original transaction. Your application processes a restore transaction by retrieving the original transaction and using it to unlock the purchased content. After Store Kit restores all the previous transactions, it notifies the payment queue observers by calling their `paymentQueueRestoreCompletedTransactionsFinished:` method.

If the user attempts to purchase a restorable product (instead of using the restore interface you implemented), the application receives a regular transaction for that item, not a restore transaction. However, the user is not charged again for that product. Your application should treat these transactions identically to those of the original transaction.

Non-renewing subscriptions and consumable products are not automatically restored by Store Kit. Non-renewing subscriptions must be restorable, however. To restore these products, you must record transactions on your own server when they are purchased and provide your own mechanism to restore those transactions to the user's devices.

Adding a Store to Your Application

This chapter provides guided steps for adding a store to your application.

The Step-By-Step Process

When you set up the project, make sure to link to `StoreKit.framework`. You can then add a store by following these steps:

1. Decide what products you wish to deliver with your application.

   There are limitations on the types of features you can offer. Store Kit does not allow your application to patch itself or download additional code. Products must either work with existing code in your application or must be implemented using data files delivered from a remote server. If you wish to add a feature that requires changes to your source code, you need to ship an updated version of your application.

2. Register product information for each product with iTunes Connect.

   You revisit this step every time you want to add a new product to your application's store. Every product requires a unique product identifier string. The App Store uses this string to look up product information and to process payments. Product identifiers are specific to your iTunes Connect account and are registered with iTunes Connect in a way similar to how you registered your application.

   The process to create and register product information is described in *iTunes Connect Developer Guide*.

3. Determine whether payments can be processed.

   A user can disable the ability to make purchases inside applications. Your application should check to see whether payments can be purchased before queuing new payment requests. Your application might do this before displaying a store to the user (as shown here) or it may defer this check until the user actually attempts to purchase an item. The latter allows the user to see items that they could purchase when payments are enabled.

   ```
   if ([SKPaymentQueue canMakePayments]) {
      // Display a store to the user.
   } else {
      // Warn the user that purchases are disabled.
   }
   ```

4. Retrieve information about products.

   Your application creates an `SKProductsRequest` object and initializes it with a set of product identifiers for the items you wish to sell, attaches a delegate to the request, and then starts it. The response holds the localized product information for all valid product identifiers.

   Keep the array of valid products from the response. When the user selects a product to buy, you will need a product object to create the payment request.

```
- (void) requestProductData
{
    SKProductsRequest *request= [[SKProductsRequest alloc]
initWithProductIdentifiers:
            [NSSet setWithObject: kMyFeatureIdentifier]];
    request.delegate = self;
    [request start];
}
- (void)productsRequest:(SKProductsRequest *)request
didReceiveResponse:(SKProductsResponse *)response
{
    NSArray *myProducts = response.products;
    // Populate your UI from the products list.
    // Save a reference to the products list.
}
```

5. Add a user interface that displays products to the user.

   Store Kit does not provide user interface classes. The look and feel of how you offer products to your customers is up to you!

6. Register a transaction observer with the payment queue.

   Your application should instantiate a transaction observer and add it as an observer of the payment queue.

```
MyStoreObserver *observer = [[MyStoreObserver alloc] init];
[[SKPaymentQueue defaultQueue] addTransactionObserver:observer];
```

   Your application should add the observer when your application launches. The App Store remembers queued transactions even if your application exited before completing all transactions. Adding an observer during initialization ensures that all previously queued transactions are seen by your application.

7. Implement the `paymentQueue:updatedTransactions:` method on `MyStoreObserver`.

The observer's `paymentQueue:updatedTransactions:` method is called whenever new transactions are created or updated.

```
- (void)paymentQueue:(SKPaymentQueue *)queue updatedTransactions:(NSArray
 *)transactions
{
    for (SKPaymentTransaction *transaction in transactions)
    {
        switch (transaction.transactionState)
        {
            case SKPaymentTransactionStatePurchased:
                [self completeTransaction:transaction];
                break;
            case SKPaymentTransactionStateFailed:
                [self failedTransaction:transaction];
                break;
            case SKPaymentTransactionStateRestored:
                [self restoreTransaction:transaction];
            default:
                break;
        }
    }
}
```

8. Your observer provides the product when the user successfully purchases an item.

```
- (void) completeTransaction: (SKPaymentTransaction *)transaction
{
    // Your application should implement these two methods.
    [self recordTransaction:transaction];
    [self provideContent:transaction.payment.productIdentifier];

    // Remove the transaction from the payment queue.
    [[SKPaymentQueue defaultQueue] finishTransaction: transaction];
}
```

A successful transaction includes a `transactionIdentifier` property and a `transactionReceipt` property that record the details of the processed payment. Your application is not required to do anything with this information. You may wish to record this information to establish an audit trail for the transaction. If your application uses a server to deliver content, the receipt can be sent to your server and validated by the App Store.

It is critical that your application take whatever steps are necessary to provide the product that the user purchased. Payment has already been collected, so the user expects to receive the new purchase. See "Feature Delivery" (page 9) for suggestions on how you might implement this.

Once you've delivered the product, your application must call `finishTransaction:` to complete the transaction. When you call `finishTransaction:`, the transaction is removed from the payment queue. To ensure that products are not lost, your application should deliver the product before calling `finishTransaction:`.

9. Finish the transaction for a restored purchase.

```
- (void) restoreTransaction: (SKPaymentTransaction *)transaction
{
    [self recordTransaction: transaction];
    [self provideContent:
transaction.originalTransaction.payment.productIdentifier];
    [[SKPaymentQueue defaultQueue] finishTransaction: transaction];
}
```

This routine is similar to that for a purchased item. A restored purchase provides a new transaction, including a different transaction identifier and receipt. You can save this information separately as part of any audit trail if you desire. However, when it comes time to complete the transaction, you'll want to recover the original transaction that holds the actual payment object and use its product identifier.

10. Finish the transaction for a failed purchase.

```
- (void) failedTransaction: (SKPaymentTransaction *)transaction
{
    if (transaction.error.code != SKErrorPaymentCancelled) {
        // Optionally, display an error here.
    }
    [[SKPaymentQueue defaultQueue] finishTransaction: transaction];
}
```

Usually a transaction fails because the user decided not to purchase the item. Your application can read the `error` field on a failed transaction to learn exactly why the transaction failed.

The only requirement for a failed purchase is that your application remove it from the queue. If your application chooses to put up an dialog displaying the error to the user, you should avoid presenting an error when the user cancels a purchase.

11. With all the infrastructure in place, you can finish the user interface. When the user selects an item in the store, create a payment object and add it to the payment queue.

```
SKProduct *selectedProduct = <#from the products response list#>;
SKPayment *payment = [SKPayment paymentWithProduct:selectedProduct];
[[SKPaymentQueue defaultQueue] addPayment:payment];
```

If your store offers the ability to purchase more than one of a product, you can create a single payment and set the quantity property.

```
SKProduct *selectedProduct = <#from the products response list#>;
SKMutablePayment *payment = [SKMutablePayment
paymentWithProduct:selectedProduct];
payment.quantity = 3;
[[SKPaymentQueue defaultQueue] addPayment:payment];
```

Where to Go Next

The code provided in these steps is best used for the built-in product model. If your application uses a server to deliver content, you are responsible for designing and implementing the protocols used to communicate between your application and your server. Your server should also verify receipts before delivering products to your application.

Verifying Store Receipts

Important: There is a vulnerability in iOS 5.1 and earlier related to receipt validation. For more details and a mitigation strategy, see *In-App Purchase Receipt Validation on iOS*.

Your application should perform the additional step of verifying that the receipt you received from Store Kit came from Apple. This is particularly important when your application relies on a separate server to provide subscriptions, services, or downloadable content. Verifying receipts on your server ensures that requests from your application are valid.

Note: On iOS, the contents and format of the store receipt is private and subject to change. Your application should not attempt to parse the receipt data directly. Use the mechanism described here to validate the receipt and retrieve the information stored inside it.

On OS X, the contents and format of the store receipt are described in *Validating Mac App Store Receipts*. OS X supports both the server validation method described in this chapter and the local validation method described in *Validating Mac App Store Receipts*.

Verifying a Receipt with the App Store

When Store Kit returns a completed purchase to your payment queue observer, the transaction's `transactionReceipt` property contains a signed receipt that records all the critical information for the transaction. Your server can post this receipt to the App Store to verify that the receipt is valid and has not been tampered with. Queries transmitted directly to the App Store are sent and received as JSON dictionaries, as defined in RFC 4627.

To verify the receipt, perform the following steps:

1. Retrieve the receipt data. On iOS, this is the value of the transaction's `transactionReceipt` property. On OS X, this is the entire contents of the receipt file inside the application bundle. Encode the receipt data using base64 encoding.
2. Create a JSON object with a single key named `receipt-data` and the string you created in step 1. Your JSON code should look like this:

```
{
    "receipt-data" : "(receipt bytes here)"
}
```

3. Post the JSON object to the App Store using an HTTP POST request. The URL for the store is `https://buy.itunes.apple.com/verifyReceipt`.
4. The response received from the App Store is a JSON object with two keys, `status` and `receipt`. It should look something like this:

```
{
    "status" : 0,
    "receipt" : { (receipt here) }
}
```

If the value of the `status` key is `0`, this is a valid receipt. If the value is anything other than `0`, this receipt is invalid.

The Store Receipt

The receipt data you send to the App Store encodes information about the transaction. When the App Store validates a receipt, the data stored in the receipt data are decoded and returned in the `receipt` key of the response. The receipt response is a JSON dictionary that includes all of the information returned to your application in the `SKPaymentTransaction` object. Your server can query these fields to retrieve the details of the purchase. Apple recommends that you send only the receipt data to your server and use receipt validation to retrieve the purchase details. Because the App Store verifies that the receipt data has not been tampered with, retrieving this information from the response is more secure than transmitting both receipt data and the transaction data to your server.

Table 5-1 provides a list of keys that you may use to retrieve information about the purchase. Many of these keys match properties on the `SKPaymentTransaction` class. All keys not specified in Table 5-1 are reserved for Apple.

Note: Some keys vary depending on whether your application is connected to the App Store or the sandbox testing environment. For more information on the sandbox, see "Testing a Store" (page 28).

Table 5-1 Purchase info keys

Key	Description
`quantity`	The number of items purchased. This value corresponds to the `quantity` property of the `SKPayment` object stored in the transaction's `payment` property.
`product_id`	The product identifier of the item that was purchased. This value corresponds to the `productIdentifier` property of the `SKPayment` object stored in the transaction's `payment` property.
`transaction_id`	The transaction identifier of the item that was purchased. This value corresponds to the transaction's `transactionIdentifier` property.
`purchase_date`	The date and time this transaction occurred. This value corresponds to the transaction's `transactionDate` property.
`original_transaction_id`	For a transaction that restores a previous transaction, this holds the original transaction identifier.
`original_purchase_date`	For a transaction that restores a previous transaction, this holds the original purchase date.
`app_item_id`	A string that the App Store uses to uniquely identify the application that created the payment transaction. If your server supports multiple applications, you can use this value to differentiate between them. Applications that are executing in the sandbox do not yet have an `app-item-id` assigned to them, so this key is missing from receipts created by the sandbox.
`version_external_identifier`	An arbitrary number that uniquely identifies a revision of your application. This key is missing in receipts created by the sandbox.
`bid`	The bundle identifier for the application.
`bvrs`	A version number for the application.

Testing a Store

During development, you should test your application to ensure that purchases are working correctly. However, you do not want to charge users while testing your application. Apple provides a sandbox environment to allow you to test your application without creating financial transactions.

> **Note:** Store Kit can be tested in the iOS Simulator, except for hosted content downloads.

The Sandbox Environment

When you launch your application from Xcode, Store Kit does not connect to the App Store. Instead, it connects to a special sandbox store environment. The sandbox environment uses the infrastructure of the App Store, but it does not process actual payments. It returns transactions as if payments were processed successfully. The sandbox environment uses special iTunes Connect accounts that are limited to In-App Purchase testing. You cannot use your normal iTunes Connect account to test your store in the sandbox.

To test your application, create one or more special test accounts in iTunes Connect. You should make at least one test account for each region that your application is localized for. Detailed information about creating test accounts can be found in *iTunes Connect Developer Guide*.

Testing in the Sandbox

Follow these steps to test your application in the sandbox.

1. Log out from your iTunes account on the test device.

   Before you can test your application, you must first log out of your regular iTunes account. iOS 3.0 includes a Store category in the Settings application. To log out of your iTunes account, exit your application, launch the Settings application and click the Store icon. Sign out from the currently active account.

   > **Important:** Do not sign in with your test account in the Settings application.

2. Launch your application.

Once you have signed out of your account, exit Settings and launch your application. As you make purchases from your application's store, Store Kit prompts you to authenticate the transaction. Log in using your test account to approve the payment. No financial transaction takes place, but transactions complete as if a payment was made.

Validating Receipts in the Sandbox

You may also validate receipts created by the sandbox environment. The code to validate a receipt received from the sandbox is identical to that for the regular App Store, except your server must direct the request to the sandbox URL.

```
NSURL *sandboxStoreURL = [[NSURL alloc] initWithString:
@"https://sandbox.itunes.apple.com/verifyReceipt"];
```

Auto-Renewable Subscriptions

Note: Subscriptions are available only on iOS.

In-App Purchase provides a standardized way to implement auto-renewable subscriptions. Auto-renewable subscriptions have a few notable characteristics:

- When you configure an auto-renewable subscription in iTunes Connect, you also configure the duration for the subscription and other marketing options.
- Auto-renewable subscriptions are restored automatically using the same Store Kit functions used to restore non-consumable products. The original purchase transaction as well a transaction for each renewal are sent to your application. See "Restoring Transactions" (page 19).
- When your server verifies a receipt with the App Store and the subscription is both active and has been renewed by the App Store, the App Store returns an updated receipt to your application.

Adding Auto-Renewable Subscriptions to Your Store

To implement auto-renewable subscriptions, follow these steps:

- Connect to iTunes Connect and use it to create a new shared secret. A shared secret is a password that your server must provide when validating receipts for auto-renewable subscriptions. This shared secret provides an additional layer of security to your transactions with the App Store. See *iTunes Connect Developer Guide*.
- Configure new products in iTunes Connect with the new auto-renewable subscription type.
- Modify your server's receipt verification code so that it adds the shared secret to the JSON data it sends to the App Store. Your server's verification code must parse the response to determine whether the subscription has expired. If the subscription has been renewed by the user, the most current receipt is also returned to your server.

Designing your Client Application

In most cases, your client application should require minimal changes to support auto-renewable subscriptions. In fact, your client application is now made simpler, as you can use the same code to recover auto-renewable subscriptions as you do to recover non-consumable products. Your application receives a separate transaction for each period of time where the subscription was renewed; your application should verify each receipt separately.

Verifying an Auto-renewable Subscription Receipt

Verifying a receipt for an auto-renewable subscription is almost identical to the process described in "Verifying Store Receipts" (page 25). Your application creates a JSON object and posts it to the App Store. The JSON object for an auto-renewable subscription receipt must include a second parameter — the shared secret you created earlier on iTunes Connect:

```
{
    "receipt-data" : "(receipt bytes here)",
    "password"     : "(shared secret bytes here)"
}
```

The response includes a status field that indicates whether the receipt was successfully validated.

```
{
    "status" : 0,
    "receipt" : { (receipt here) },
    "latest_receipt" : "(base-64 encoded receipt here)",
    "latest_receipt_info" : { (latest receipt info here) }
}
```

If the user's receipt was valid and the subscription is active, the status field holds `0`, and the `receipt` field is populated with the decoded receipt data. If your server receives a non-zero status value, use Table 7-1 to interpret non-zero status codes.

Table 7-1 Status codes for auto-renewable subscriptions

Status Code	Description
21000	The App Store could not read the JSON object you provided.
21002	The data in the receipt-data property was malformed.
21003	The receipt could not be authenticated.
21004	The shared secret you provided does not match the shared secret on file for your account.
21005	The receipt server is not currently available.
21006	This receipt is valid but the subscription has expired. When this status code is returned to your server, the receipt data is also decoded and returned as part of the response.
21007	This receipt is a sandbox receipt, but it was sent to the production service for verification.
21008	This receipt is a production receipt, but it was sent to the sandbox service for verification.

Important: The non-zero status codes here apply only when recovering information about a auto-renewable subscription. Do not use these status codes when testing responses for other kinds of products.

The receipt field on the JSON object holds the parsed information from the receipt. The receipt data for an auto-renewable subscription includes one additional key, and some other key previously described in Table 5-1 (page 27) are modified slightly for subscriptions. See Table 7-2 for details on the new and modified keys.

Table 7-2 Auto-renewable subscription info keys

Key	Description
`expires_date`	The expiration date of the subscription receipt, expressed as the number of milliseconds since January 1, 1970, 00:00:00 GMT.
`original_transaction_id`	This holds the transaction identifier for the initial purchase. All subsequent renewals of this subscription and recovered transactions all share this identifier.
`original_purchase_date`	This holds the purchase date for the initial purchase; it represents the start date for the subscription.

Key	Description
`purchase_date`	This holds the billing date when this transaction occurred. For a transaction for a renewable subscription, this would be the date when the subscription was renewed. If the receipt being parsed by the App Store was the latest receipt for this subscription, this field holds the date when this subscription was most recently renewed.

In addition to the `receipt_data` field, the response may also include two new fields. If the user's subscription is active and was renewed by a transaction that took place after the receipt your server sent to the App Store, the `latest_receipt` field includes a base-64 encoded receipt for the last renewal for this subscription. The decoded data for this new receipt is also provided in the `latest_expired_receipt_info` field. Your server can use this new receipt to maintain a record of the most recent renewal.

Document Revision History

This table describes the changes to *In-App Purchase Programming Guide*.

Date	Notes
2012-09-19	Removed note that expires_date key was not present on restored transactions. Best practice for restoring auto-renewable subscriptions is to simply respect the expires_date key on the restored transactions. Removed section on restoring auto-renewable subscriptions that indicated otherwise.
2012-02-16	Updated artwork throughout to reflect cross-platform availability. Updated code listing to remove deprecated method. Replaced the deprecated `paymentWithProductIdentifier:` method with the `paymentWithProduct:` method in "Adding a Store to Your Application" (page 20).
2012-01-09	Minor updates for using this technology on OS X.
2011-10-12	Added information about a new type of purchase to the overview.
2011-06-06	First release of this document for OS X.
2011-05-26	Updated to reflect the latest server behavior for auto-renewable subscriptions.
2011-03-08	Corrected the list of keys in the renewable subscriptions chapter.
2011-02-15	Apps must always retrieve product information before submitting a purchase; this ensures that the item has been marked for sale. Information added about renewable subscriptions.
2010-09-01	Minor edits.

Date	Notes
2010-06-14	Minor clarifications to SKRequest.
2010-01-20	Fixed a typo in the JSON receipt object.
2009-11-12	Receipt data must be base64 encoded before being passed to the validation server. Other minor updates.
2009-09-09	Revised introductory chapter. Clarified usage of receipt data. Recommended the iTunes Connect Developer Guide as the primary source of information about creating product metadata. Renamed from "Store Kit Programming Guide" to "In App Purchase Programming Guide".
2009-06-12	Revised to include discussion on retrieving price information from the Apple App Store as well as validating receipts with the store.
2009-03-13	New document that describes how to use the StoreKit API to implement a store with your application.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010