Article

Downloading Apple-hosted asset packs

Configure your project and write the code to download asset packs hosted by Apple.

Overview

Let Apple host your app's assets for you with your Apple Developer Program membership. Apple-Hosted Background Assets hosts up to 200GB of compressed assets and is available for apps on TestFlight or the App Store that use the Managed Background Assets features on all platforms except watchOS. You can use the service to host many different types of assets — such as texture files, machine learning models, Metal shader libraries, and videos — and update them independent of your app build. Background Assets provides a default implementation of a downloader extension that manages asset pack downloads, background updates, compression, and more on people's systems that you can customize in your downloader extension code.

To create asset packs with a manifest file that the system manages, see Creating managed asset packs. To test your asset packs before uploading them to App Store Connect, see Testing asset packs locally.

Configure your project for Apple-Hosted Background Assets

To use Apple-Hosted Background Assets, add a downloader extension target to your project, configure the App Groups capability for both the app and the extension targets, and add some Background Assets information property list keys.

Background Assets uses your app's downloader extension to schedule downloads when your app isn't running. Add a downloader extension that is preconfigured for Managed Background Assets to your Xcode project from a template. Choose New > Target and, in the sheet that appears, choose the Background Download template under Application Extension, and click Next. In the dialog, enter a target name, choose Apple-Hosted, Managed as the extension type, and click Finish. In the next dialog, click Activate to use the extension scheme Xcode creates.

Your app and your downloader extension target need to share an app group, which Background Assets uses to facilitate coordination between the two. Add each of the two targets to a shared app group in the Signing & Capabilities tab of the target editor in Xcode.

In the project editor, select the app target and click the Info tab. Then, add the following keys to the information property list file:

`BAAppGroupID`
: The string ID of the app group that your app and downloader extension targets share.

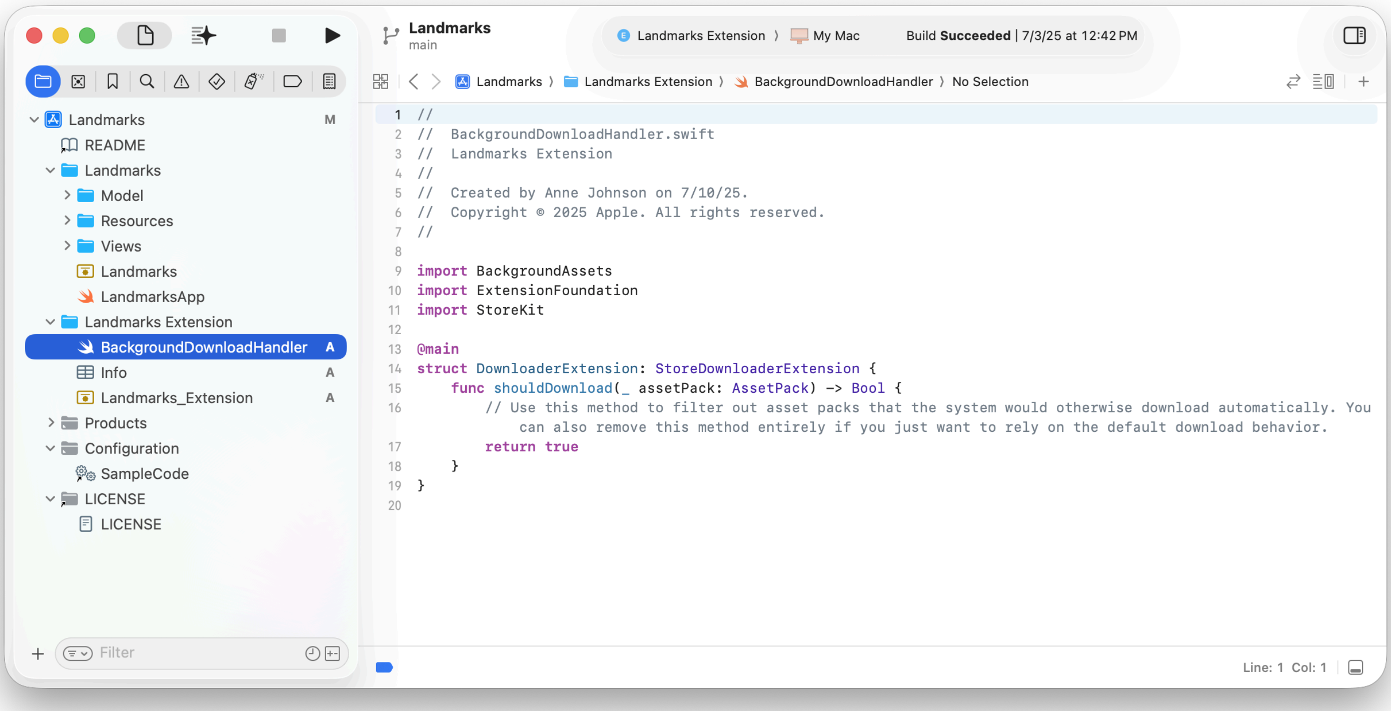
`BAHasManagedAssetPacks`
: For apps that use Managed Background Assets (including Apple-Hosted Background Assets), you must set this key to `YES`.

`BAUsesAppleHosting`
: For apps that use Apple-Hosted Background Assets, you must set this key to `YES`.

For apps that use Apple-Hosted Background Assets, omit all other Background Assets information property list keys from your project.

Customize downloads

The default system implementation supports automatic downloads, background updates, compression, and more that you can customize. The system calls the `shouldDownload(_:)` method each time it downloads a new asset pack based on its download policy. Optionally, use this method to filter downloads at runtime.



```swift
//
//  BackgroundDownloadHandler.swift
//  Landmarks Extension
//
//  Created by Anne Johnson on 7/10/25.
//  Copyright © 2025 Apple. All rights reserved.
//

import BackgroundAssets
import ExtensionFoundation
import StoreKit

@main
struct DownloaderExtension: StoreDownloaderExtension {
    func shouldDownload(_ assetPack: AssetPack) -> Bool {
        // Use this method to filter out asset packs that the system would otherwise download automatically. You
            can also remove this method entirely if you just want to rely on the default download behavior.
        return true
    }
}
```

If you don't need to customize the download behavior for your asset packs beyond the download policies that you configure in the manifest files, remove the `shouldDownload(_:)` method implementation from your downloader-extension structure. Otherwise, if your asset packs have specific compatibility requirements, provide a custom implementation for the `shouldDownload(_:)` method.

Download asset packs from your app

An instance of the `AssetPack` structure represents an individual asset pack that's available for download from the server, including metadata, such as its identifier that you set in the manifest file. To obtain an `AssetPack` instance, call the `assetPack(withID:)` method on the shared asset-pack manager:

```swift
let assetPack = try await AssetPackManager.shared.assetPack(withID: "Tutorial")
```

To ensure that the system downloads an asset pack, pass it to the `ensureLocalAvailability(of:)` method:

```swift
try await AssetPackManager.shared.ensureLocalAvailability(of: assetPack)
```

If the system previously downloaded the asset pack, the `ensureLocalAvailability(of:)` method returns quickly without downloading it again. For an asset pack with an essential or a prefetch download policy, the system may finish downloading the asset pack before you call this method. To download an asset pack with an on-demand download policy, this method initiates the download and waits for it to finish before returning. If this method returns without throwing an error, the asset pack is available to access locally.

> Note
>
> Even for an asset pack with an essential download policy, network dropouts or other uncommon issues can prevent the system from downloading it until you call the `ensureLocalAvailability(of:)` method.

Show download progress

When downloading asset packs in the foreground, display a progress indicator.

In Swift, you can await status updates on the asynchronous sequence that the `statusUpdates(forAssetPackWithID:)` method returns:

```
let statusUpdates = AssetPackManager.shared.statusUpdates(forAssetPackWithID: "Tuto
for await statusUpdate in statusUpdates {
    switch statusUpdate {
    case began(let assetPack): // …
    case paused(let assetPack): // …
    case downloading(let assetPack, let progress): // …
    case finished(let assetPack): // …
    case failed(let assetPack, let error): // …
    }
}
```

In Objective-C, you can create a class that conforms to the `BAManagedAssetPackDownloadDelegate` protocol:

```
@interface ManagedAssetPackDownloadDelegate : NSObject <BAManagedAssetPackDownloadDe

@end

@implementation ManagedAssetPackDownloadDelegate

- (void)downloadOfAssetPackBegan:(BAAssetPack*)assetPack { /* … */ }

- (void)downloadOfAssetPackPaused:(BAAssetPack*)assetPack { /* … */ }
- (void)downloadOfAssetPack:(BAAssetPack*)assetPack hasProgress:(NSProgress*)progres

- (void)downloadOfAssetPackFinished:(BAAssetPack*)assetPack { /* … */ }

- (void)downloadOfAssetPack:(BAAssetPack*)assetPack failedWithError:(NSError*)error

@end
```

Then, set the shared asset-pack manager's `delegate` property to an instance of that class.

```
ManagedAssetPackDownloadDelegate* delegate = [[ManagedAssetPackDownloadDelegate allo
[[BAAssetPackManager sharedManager] setDelegate:delegate];
```

To cancel a download, call the `cancel()` method on any of the `Progress` objects that you receive in the download status updates:

```
switch statusUpdate {
case downloading(let assetPack, let progress) where assetPack.id == "Tutorial":
    progress.cancel()
default: // …
}
```

Load files in a downloaded asset pack

To read the contents of an asset pack, call the `contents(at:searchingInAssetPackWithID:options:)` method on the shared asset-pack manager, passing a relative path from the root of the source repository — the folder where you ran the packaging tool command — to the file that you want to read:

```
let videoData = try AssetPackManager.shared.contents(at: "Videos/Introduction.m4v")
```

The system automatically merges all of your asset packs into a shared namespace, effectively reconstructing your asset root folder as if it were pasted on a person's device. This way, you can access individual files without needing to know which asset pack they reside in.

By default, the `contents(at:searchingInAssetPackWithID:options:)` method returns a memory-mapped `Data` instance, which is suitable even for large asset files that take up a lot of space in memory. If you need low-level access to the file descriptor — for example, to read a file into memory procedurally — then you can use the `descriptor(for:searchingInAssetPackWithID:)` method instead:

```
let videoDescriptor = try AssetPackManager.shared.descriptor(for: "Videos/Introducti
defer {
    do {
        try videoDescriptor.close()
    } catch {
        print("The file descriptor couldn’t be closed: \(error)")
    }
}
```

Important

It’s your responsibility to close the file descriptor when you’re done using it.

The system tracks which asset packs you download and automatically keeps them up to date in the background. However, the system won’t automatically remove your asset packs while your app is installed. Therefore, when you are done with an asset pack, call the `remove(assetPackWithID:)` method on the shared asset-pack manager to free up storage space on the device. For example, remove the tutorial asset pack when a person finishes playing the tutorial:

```
try await AssetPackManager.shared.remove(assetPackWithID: "Tutorial")
```

To redownload an asset pack, call the `ensureLocalAvailability(of:)` method again.

See Also

Essentials

Creating managed asset packs

Create managed asset packs, choose download options, and upload Apple-hosted asset packs to App Store Connect.

Testing asset packs locally

Test your system-managed asset packs using a mock server on your Mac.