Sample Code

Using AVFoundation to play and persist HTTP live streams

Play HTTP Live Streams and persist streams on disk for offline playback using AVFoundation.

Download

iOS 18.0+ | iPadOS 18.0+ | Xcode 16.0+

Overview

This sample provides a catalog of HTTP Live Streams (HLS) that you can play by tapping the row in the table corresponding to the stream. To manage the download of a stream, tap the button associated with the stream in the table. Tapping the button causes a transition to a new view controller which provides an interface to initiate a download, cancel an already running download, or delete a downloaded stream from the device.

The sample creates and initializes an `AVAssetDownloadConfiguration` and creates a `AVAssetDownloadTask` using the download configuration for the download of a stream. The example shows how to set a primary `AVAssetDownloadContentConfiguration` and at least one auxiliary content configuration to be downloaded.

> **Note**
>
> This sample doesn't support saving FairPlay Streaming (FPS) content. For a version of the sample that demonstrates how to download FPS content, see FairPlay Streaming Server SDK.

Configure the sample code project

Build and run the sample on an actual device or a simulator device running iOS 15 or later.

If you want to add your own streams to test with this sample, add an entry into the `Streams.plist` file in the Xcode project. There are two important keys you need to provide values for:

`name`
: The display name of the HLS stream in the sample.

`playlist_url`
: The URL of the HLS stream's master playlist.

If any of the streams you add aren't hosted securely, you'll need to add an Application Transport Security (ATS) exception in the `Info.plist` file in the Xcode project. For more information on ATS and the relevant property list keys see `NSAppTransportSecurity`

Play a stream

To play an item, tap one of the rows in the table. Tapping the item causes a transition to a new view controller. As part of that transition, the table view creates an `AssetPlaybackManager` and assigns the appropriate asset to it, as shown in the following example:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    super.prepare(for: segue, sender: sender)

    if segue.identifier == AssetListTableViewController.presentPlayerViewControllerS
        guard let cell = sender as? AssetListTableViewCell,
            let playerViewControler = segue.destination as? AVPlayerViewController e

        /*
         Grab a reference for the destinationViewController to use in later delegate
         AssetPlaybackManager.
         */
        playerViewController = playerViewControler

        // Load the new Asset to playback into AssetPlaybackManager.
        AssetPlaybackManager.sharedManager.setAssetForPlayback(cell.asset)
    }
}
```

Assigning an asset to the `AssetPlaybackManager` causes it to create an `AVPlayerItem` for the asset, removing any previous asset in the process:

```
private var asset: Asset? {
    willSet {
        /// Remove any previous KVO observer.
        guard let urlAssetObserver = urlAssetObserver else { return }

        urlAssetObserver.invalidate()
    }

    didSet {
        if let asset {
            Task {
                do {
                    if try await asset.urlAsset.load(.isPlayable) {
                        playerItem = AVPlayerItem(asset: asset.urlAsset)
                        player.replaceCurrentItem(with: playerItem)
                    } else {
                        // The asset isn't playable, so reset the player state.
                        resetPlayer()
                    }
                } catch {
                    logger.error("Unable to load `isPlayable` property.")
                }
            }
        } else {
            resetPlayer()
        }
    }
}
```

The `AssetPlaybackManager` uses KVO to monitor the `AVPlayerItem` object's `status` and initiates playback when the `status` becomes ready to play:

```
playerItemObserver = playerItem?.observe(\AVPlayerItem.status, options: [.new, .init
    guard let strongSelf = self else { return }

    if item.status == .readyToPlay {
        if !strongSelf.readyForPlayback {
            strongSelf.readyForPlayback = true
            strongSelf.delegate?.streamPlaybackManager(strongSelf, playerReadyToPlay
        }
    } else if item.status == .failed {
        let error = item.error
```

```
        logger.error("Error: \(String(describing: error?.localizedDescription))")
    }
```

Download a stream

When the person initiates a download by tapping the button in the corresponding stream's table view cell, an instance of `AssetPersistenceManager` calls the following function to create an `AVAssetDownloadTask` object with an `AVAssetDownloadConfiguration` to download multiple `AVMediaSelection` for the `AVURLAsset` of the stream:

```
func downloadStream(for asset: Asset) async throws {

    // Get the default media selections for the asset's media selection groups.
    let preferredMediaSelection = try await asset.urlAsset.load(.preferredMediaSelec

    /*
     Creates and initializes an `AVAssetDownloadTask` using an `AVAssetDownloadConfi
     on an `AVURLAsset`.
     The `primaryContentConfiguration` in `AVAssetDownloadConfiguration` requests fo
     lower bitrate variants in the asset.
     */
    let config = AVAssetDownloadConfiguration(asset: asset.urlAsset, title: asset.st
    /// Primary content configuration setup.
    let primaryQualifier = AVAssetVariantQualifier(predicate: NSPredicate(format: "p
    config.primaryContentConfiguration.variantQualifiers = [primaryQualifier]

    /// Creation of `AVAssetDownloadTask` with the above configured `AVAssetDownload
    let task = assetDownloadURLSession.makeAssetDownloadTask(downloadConfiguration:

    /// To better track the `AVAssetDownloadTask`, set the `taskDescription` to some
    task.taskDescription = asset.stream.name

    activeDownloadsMap[task] = asset

    /// Use `task.progress` value to provide download progress updates in the UI.
    let progressObservation: NSKeyValueObservation = task.progress.observe(\.fractio
        Task { @MainActor in
            var userInfo = [String: Any]()
            userInfo[Asset.Keys.name] = asset.stream.name
            userInfo[Asset.Keys.percentDownloaded] = progress.fractionCompleted
            NotificationCenter.default.post(name: .AssetDownloadProgress, object: ni
```

```
        }
    }
    self.progressObservers.append(progressObservation)

    task.resume()

    var userInfo = [String: Any]()
    userInfo[Asset.Keys.name] = asset.stream.name
    userInfo[Asset.Keys.downloadState] = Asset.DownloadState.downloading.rawValue
    userInfo[Asset.Keys.downloadSelectionDisplayName] = await displayNamesForSelecte

    NotificationCenter.default.post(name: .AssetDownloadStateChanged, object: nil, u
}
```

> **Note**
>
> You can't save an HTTP Live Stream while it's in progress. If you try to save a live stream, the system throws an exception. Only Video On Demand (VOD) streams support offline playback.

Cancel an in-progress download

Tap the button in the corresponding stream's table view cell to reveal the accessory view, then tap Cancel to stop downloading the stream. The following function in `AssetPersistenceManager` cancels the download by calling the `URLSessionTask` `cancel()` method.

```
func cancelDownload(for asset: Asset) {
    var task: AVAssetDownloadTask?

    for (taskKey, assetVal) in activeDownloadsMap where asset == assetVal {
        task = taskKey
        break
    }

    task?.cancel()
}
```

Remove a downloaded stream from disk

Tap the button in the corresponding stream's table view cell to reveal the accessory view, then tap Delete to delete the downloaded stream file. The following function in `AssetPersistence`

`Manager` removes a downloaded stream on the device. First the asset URL corresponding to the file on the device is identified, then the `FileManager` `removeItem(at:)` method is called to remove the downloaded stream at the specified URL.

```
func deleteAsset(_ asset: Asset) {
    let userDefaults = UserDefaults.standard

    do {
        if let localFileLocation = localAssetForStream(withName: asset.stream.name)?
            try FileManager.default.removeItem(at: localFileLocation)

            userDefaults.removeObject(forKey: asset.stream.name)

            var userInfo = [String: Any]()
            userInfo[Asset.Keys.name] = asset.stream.name
            userInfo[Asset.Keys.downloadState] = Asset.DownloadState.notDownloaded.r

            NotificationCenter.default.post(name: .AssetDownloadStateChanged, object
                                            userInfo: userInfo)
        }
    } catch {
        logger.error("An error occured deleting the file: \(error)")
    }
}
```

Measure playback performance

> **Note**
>
> You can view the various performance indicators in the console during playback.

For example, here's the code to calculate the total time spent playing the stream, obtained from the `AVPlayerItemAccessLog`:

```
var totalDurationWatched: Double {
    // Compute total duration watched by iterating through the AccessLog events.
    var totalDurationWatched = 0.0
    if accessLog != nil && !accessLog!.events.isEmpty {
        for event in accessLog!.events where event.durationWatched > 0 {
                totalDurationWatched += event.durationWatched
```

```
            }
        }
        return totalDurationWatched
    }
```

See Also

Asset downloading

`class AVAssetDownloadURLSession`

A URL session that creates and executes asset download tasks.

`class AVAssetDownloadTask`

A session used to download HTTP Live Streaming assets.

`class AVAggregateAssetDownloadTask`

A task that downloads multiple media selections for an asset.