Refining The User Experience

You've seen throughout this guide the ways you can use AVKit and AVFoundation to build compelling playback apps. This chapter describes some additional features of AVFoundation that you can use to further customize and refine your app's playback experience.

Presenting Chapter Markers

Chapter markers enable users to quickly navigate your content. `AVPlayerViewController` in tvOS and macOS automatically presents a chapter selection-interface if markers are found in the currently played asset. You can also directly retrieve this data whenever you want to create your own custom chapter-selection interface.

Chapter markers are a type of *timed metadata*. This is the same kind of metadata as discussed in other sections of this guide, but instead of applying to the whole asset, it applies only to ranges of time within the asset's timeline. You retrieve an asset's chapter metadata using either the `chapterMetadataGroupsBestMatchingPreferredLanguages:` or `chapterMetadataGroupsWithTitleLocale:containingItemsWithCommonKeys:` methods. These methods become callable without blocking after you asynchronously load the value of the asset's `availableChapterLocales` key:

```
let asset = AVAsset(url: <# Asset URL #>)
let chapterLocalesKey = "availableChapterLocales"

asset.loadValuesAsynchronously(forKeys: [chapterLocalesKey]) {
    var error: NSError?
    let status = asset.statusOfValue(forKey: chapterLocalesKey, error: &error)
    if status == .loaded {
        let languages = Locale.preferredLanguages
        let chapterMetadata =
asset.chapterMetadataGroups(bestMatchingPreferredLanguages: languages)
        // Process chapter metadata
    }
    else {
        // Handle other status cases
    }
}
```

The value returned from these methods is an array of `AVTimedMetadataGroup` objects, each representing an individual chapter marker. An `AVTimedMetadataGroup` object is composed of a `CMTimeRange`, defining the time range to which its metadata applies, an array of `AVMetadataItem` objects representing the chapter's title, and optionally, its thumbnail image. The following example shows how to convert the `AVTimedMetadataGroup` data into an array of custom model objects, called `Chapter`, to be presented in the app's view layer:

```
func convertTimedMetadataGroupsToChapters(groups: [AVTimedMetadataGroup]) ->
[Chapter] {
    return groups.map { group -> Chapter in

        // Retrieve AVMetadataCommonIdentifierTitle metadata items
        let titleItems =
            AVMetadataItem.metadataItems(from: group.items,
                                             filteredByIdentifier:
AVMetadataCommonIdentifierTitle)

        // Retrieve AVMetadataCommonIdentifierTitle metadata items
        let artworkItems =
            AVMetadataItem.metadataItems(from: group.items,
                                             filteredByIdentifier:
AVMetadataCommonIdentifierArtwork)

        var title = "Default Title"
        var image = UIImage(named: "placeholder")!

        if let titleValue = titleItems.first?.stringValue {
            title = titleValue
        }

        if let imgData = artworkItems.first?.dataValue, imageValue = UIImage(data:
imgData) {
            image = imageValue
        }

        return Chapter(time: group.timeRange.start, title: title, image: image)
    }
}
```

With the relevant data converted, you can build a chapter-selection interface and use the `time` value of the chapter object to seek the current presentation using the player's `seekToTime:` method.

Selecting Media Options

As a developer, you want to make your apps accessible to as broad an audience as possible. One way for you to extend your app's reach is to make it available to users in their native language as well as to provide support for users who have have hearing impairments or other accessibility needs. AVKit and AVFoundation make it easy to handle these concerns by providing built-in support for presenting subtitles and closed captions, and for selecting alternative audio and video tracks. If you're building your own custom player or would like to present your own media-selection interface, you can use the features provided by AVFoundation's `AVMediaSelectionGroup` and `AVMediaSelectionOption` classes.

An `AVMediaSelectionOption` models an alternative audio, video, or text track contained within the source media. Media options are used to select alternative camera angles, present audio dubbed in a user's native language, or display subtitles and closed captions. You determine which alternative media presentations are available by asynchronously loading and calling the asset's `availableMediaCharacteristicsWithMediaSelectionOptions` property, which returns an array of strings indicating the available media characteristics. The possible values returned are `AVMediaCharacteristicAudible` (alternative audio tracks), `AVMediaCharacteristicVisual` (alternative video tracks), and `AVMediaCharacteristicLegible` (subtitles and closed captions).

After you've retrieved the array of available options, you call the asset's `mediaSelectionGroupForMediaCharacteristic:` method, passing it the desired characteristic. This method returns the associated `AVMediaSelectionGroup` object, or `nil` if no groups exist for the specified characteristic.

AVMediaSelectionGroup acts as a container for a collection of mutually exclusive AVMediaSelectionOption objects. The following example shows how you retrieve an asset's media-selection groups and display their available options:

```
for characteristic in asset.availableMediaCharacteristicsWithMediaSelectionOptions {


    print("\(characteristic)")


    // Retrieve the AVMediaSelectionGroup for the specified characteristic.
    if let group = asset.mediaSelectionGroup(forMediaCharacteristic: characteristic)
{
        // Print its options.
        for option in group.options {
            print("  Option: \(option.displayName)")
        }
    }
}
```

The output for an asset containing audio and subtitle media options looks similar to the following:

```
[AVMediaCharacteristicAudible]
  Option: English
  Option: Spanish
[AVMediaCharacteristicLegible]
  Option: English
  Option: German
  Option: Spanish
  Option: French
```

After you've retrieved an AVMediaSelectionGroup object for a particular media characteristic and identified the desired AVMediaSelectionOption object, the next step is to select it. You select a media option by calling selectMediaOption:inMediaSelectionGroup: on the active AVPlayerItem. For instance, to present the asset's Spanish subtitle option, you could select it as follows:

```
if let group = asset.mediaSelectionGroup(forMediaCharacteristic:
AVMediaCharacteristicLegible) {
    let locale = Locale(identifier: "es-ES")
    let options =
        AVMediaSelectionGroup.mediaSelectionOptions(from: group.options, with:
locale)
    if let option = options.first {
        // Select Spanish-language subtitle option
        playerItem.select(option, in: group)
    }
}
```

Selecting a media option makes it immediately available for presentation. Selecting a subtitle or closed-caption option displays the associated text within the video display provided by AVPlayerViewController, AVPlayerView, and AVPlayerLayer. Selecting an alternative audio or video option replaces the currently presented media with the new selection's media.

Note: Starting with iOS 7.0 and OS X 10.9, `AVPlayer` provides automatic media selection based on the user's system preferences as its default behavior. To take control over when media selections are presented, disable the default behavior by setting the player's `appliesMediaSelectionCriteriaAutomatically` value to `NO`.

Working with the iOS Audio Environment

You use the iOS audio session APIs to define your application's general audio behavior and its role within the overall audio context of the device it's running on. The following sections describe additional ways to manage and control your app's audio playback and how to respond to changes in the larger iOS audio environment.

Playing Background Audio

A common feature of many media playback apps is to continue playing audio when the app is sent to the background. This may be the result of a user switching applications or locking the device. To enable your app to play background audio, you begin by configuring your app's capabilities and audio session, as described in Configuring Audio Settings for iOS and tvOS.

If you're playing audio-only assets, such as MP3 or M4A files, your setup is complete and your app can play background audio. If you need to play the audio portion of a video asset, an additional step is required. If the player's current item is displaying video on the device's display, playback of the `AVPlayer` is automatically paused when the app is sent to the background. If you want to continue playing audio, you disconnect the `AVPlayer` instance from the presentation when entering the background and reconnect it when returning to the foreground as shown in the following example:

```
func applicationDidEnterBackground(_ application: UIApplication) {
    // Disconnect the AVPlayer from the presentation when entering background

    // If presenting video with AVPlayerViewController
    playerViewController.player = nil

    // If presenting video with AVPlayerLayer
    playerLayer.player = nil
}

func applicationWillEnterForeground(_ application: UIApplication) {
    // Reconnect the AVPlayer to the presentation when returning to foreground

    // If presenting video with AVPlayerViewController
    playerViewController.player = player

    // If presenting video with AVPlayerLayer
    playerLayer.player = player
}
```

Controlling Background Audio

If your app plays audio in the background, you should support remotely controlling playback both in Control Center and from the iOS Lock screen. Along with controlling playback, you should also provide meaningful information in these interfaces about what's currently playing. To implement this functionality, you use the MediaPlayer framework's `MPRemoteCommandCenter` and `MPNowPlayingInfoCenter` classes.

The `MPRemoteCommandCenter` class vends objects for handling remote-control events sent by external accessories and system transport controls. It defines a variety of commands in the form of `MPRemoteCommand` objects to which you can attach custom event handlers to control playback in your app. For instance, to remotely control your app's play and pause behaviors, you get a reference to the shared command center and provide handlers for the appropriate commands, as shown below:

```
func setupRemoteTransportControls() {
    // Get the shared MPRemoteCommandCenter
    let commandCenter = MPRemoteCommandCenter.shared()

    // Add handler for Play Command
    commandCenter.playCommand.addTarget { [unowned self] event in
        if self.player.rate == 0.0 {
            self.player.play()
            return .success
        }
        return .commandFailed
    }

    // Add handler for Pause Command
    commandCenter.pauseCommand.addTarget { [unowned self] event in
        if self.player.rate == 1.0 {
            self.player.pause()
            return .success
        }
        return .commandFailed
    }
}
```

The preceding example shows how to add handlers for the command center's `playCommand` and `pauseCommand` using the `addTargetWithHandler:` method. The callback block you give to this method requires that you return a `MPRemoteCommandHandlerStatus` value, indicating whether the command succeeded or failed.

After you've configured your remote command handlers, the next step is to provide metadata to display on the iOS Lock screen and in Control Center's transport area. You provide a dictionary of metadata using the keys defined by `MPMediaItem` and `MPNowPlayingInfoCenter` and set that dictionary on the default instance of `MPNowPlayingInfoCenter`. The following example shows you how to set a title and artwork for the currently presented media, along with playback timing values:

```
func setupNowPlaying() {
    // Define Now Playing Info
    var nowPlayingInfo = [String : Any]()
    nowPlayingInfo[MPMediaItemPropertyTitle] = "My Movie"
    if let image = UIImage(named: "lockscreen") {
        nowPlayingInfo[MPMediaItemPropertyArtwork] =
            MPMediaItemArtwork(boundsSize: image.size) { size in
                return image
        }
    }

    nowPlayingInfo[MPNowPlayingInfoPropertyElapsedPlaybackTime] =
playerItem.currentTime().seconds

    nowPlayingInfo[MPMediaItemPropertyPlaybackDuration] =
playerItem.asset.duration.seconds

    nowPlayingInfo[MPNowPlayingInfoPropertyPlaybackRate] = player.rate


    // Set the metadata

    MPNowPlayingInfoCenter.default().nowPlayingInfo = nowPlayingInfo

}
```

This example passes dynamic timing as part of the `nowPlayingInfo` dictionary, which allows you to present player timing and progress in the remote-control interfaces. When doing so, provide an up-to-date snapshot of this timing as your app enters the background. You can also update the `nowPlayingInfo` while your app is in the background, if media information or timing is changed in a significant way.

Responding to Interruptions

Interruptions are a common part of the iOS user experience. For example, consider what happens if you're watching a movie in the Videos app and you receive a phone call or FaceTime request. In this scenario, your movie's audio is quickly faded out, playback is paused, and the sound of the ringtone is faded in. If you decline the call or request, control is returned to the Videos app, and playback begins again as the movie's audio fades in. At the center of this behavior is your app's `AVAudioSession`. As interruptions begin and end, it notifies any registered observers so they can take the appropriate action. `AVPlayer` is aware of your audio session and automatically pauses and resumes playback in response to `AVAudioSession` interruption events. To observe this `AVPlayer` behavior, use key-value observing (KVO) on the player's `rate` property so you can update your user interface as the player is paused and resumed in response to interruptions.

You can also directly observe interruption notifications posted by `AVAudioSession`. This might be useful if you would like to know if playback was paused because of an interruption or another reason, such as a route change. To observe audio interruptions, you begin by registering to observe notifications of type `AVAudioSessionInterruptionNotification`.

```
func setupNotifications() {
    let notificationCenter = NotificationCenter.default
    notificationCenter.addObserver(self,
                                   selector: #selector(handleRouteChange),
                                   name:
Notification.Name.AVAudioSessionRouteChange,
                                   object: nil)
}


func handleInterruption(notification: NSNotification) {


}
```

The posted `NSNotification` contains a populated `userInfo` dictionary providing the details of the interruption. You determine the type of interruption by retrieving the `AVAudioSessionInterruptionType` from the `userInfo` dictionary. The interruption type indicates whether the interruption has begun or has ended.

```
func handleInterruption(notification: NSNotification) {

    guard let userInfo = notification.userInfo,

        let typeValue = userInfo[AVAudioSessionInterruptionTypeKey] as? UInt,

        let type = AVAudioSessionInterruptionType(rawValue: typeValue) else {

            return
    }
    if type == .began {
        // Interruption began, take appropriate actions
    }
    else if type == .ended {
        if let optionsValue = userInfo[AVAudioSessionInterruptionOptionKey] as? UInt
{
            let options = AVAudioSessionInterruptionOptions(rawValue: optionsValue)
            if options == .shouldResume {
                // Interruption Ended - playback should resume
            } else {
                // Interruption Ended - playback should NOT resume
            }
        }
    }
}
```

If the interruption type is `AVAudioSessionInterruptionTypeEnded`, the `userInfo` dictionary contains an `AVAudioSessionInterruptionOptions` value, which is used to determine if playback should automatically resume.

Responding to Route Changes

An important responsibility handled by `AVAudioSession` is managing audio route changes. A route change occurs when an audio input or output is added to or removed from an iOS device. Route changes occur for a number of reasons, including a user plugging in a pair of headphones, connecting a Bluetooth LE headset, or unplugging a USB audio interface. When these changes occur, `AVAudioSession` reroutes audio signals accordingly and broadcasts a notification containing the details of the change to any registered observers.

An important requirement related to route changes occurs when a user plugs in or removes a pair of headphones (see Sound in *iOS Human Interface Guidelines*). When users connect a pair of wired or wireless headphones, they are implicitly indicating that audio playback should continue, but privately. They expect an app that is currently playing media to continue playing without pause. When users unplug their headphones, they don’t want to automatically share what they are listening to with others. Applications should respect this implicit privacy request and automatically pause playback when headphones are removed.

`AVPlayer` is aware of your app’s audio session and responds appropriately to route changes. When headphones are connected, playback continues as expected. When headphones are removed, playback is automatically paused. To observe this `AVPlayer` behavior, use KVO on the player’s `rate` property so that you can update your user interface as the player is paused in response to an audio route change.

You can also directly observe any route change notifications posted by `AVAudioSession`. This might be useful if you want to be notified when a user connects or removes headphones so you can present an icon or message in the player interface. To observe audio route changes, you begin by registering to observe notifications of type `AVAudioSessionRouteChangeNotification`.

```
func setupNotifications() {
    let notificationCenter = NotificationCenter.default
    notificationCenter.addObserver(self,
                                   selector: #selector(handleRouteChange),
                                   name:
Notification.Name.AVAudioSessionRouteChange,
                                   object: nil)
}
```

```
func handleRouteChange(notification: NSNotification) {


}
```

The posted `NSNotification` contains a populated `userInfo` dictionary providing the details of the route change. You can determine the reason for this change by retrieving the `AVAudioSessionRouteChangeReason` from the `userInfo` dictionary. When a new device is connected, the reason is `AVAudioSessionRouteChangeReasonNewDeviceAvailable`, and when one is removed, it is `AVAudioSessionRouteChangeReasonOldDeviceUnavailable`.

```
func handleRouteChange(notification: NSNotification) {
    guard let userInfo = notification.userInfo,
        let reasonValue = userInfo[AVAudioSessionRouteChangeReasonKey] as? UInt,
        let reason = AVAudioSessionRouteChangeReason(rawValue:reasonValue) else {
            return
    }
    switch reason {
    case .newDeviceAvailable:
        // Handle new device available.
    case .oldDeviceUnavailable:
        // Handle old device removed.
    default: ()
    }
}
```

When a new device becomes available, you ask the `AVAudioSession` for its `currentRoute` to determine where the audio output is currently routed. This returns an `AVAudioSessionRouteDescription` listing all of the audio session’s `inputs` and `outputs`. When a device is removed, you retrieve the `AVAudioSessionRouteDescription` for the *previous* route from the `userInfo` dictionary. In both cases, you query the route description for its `outputs`, which returns an array of `AVAudioSessionPortDescription` objects providing the details of the audio output routes.

```
func handleRouteChange(notification: NSNotification) {
    guard let userInfo = notification.userInfo,
        let reasonValue = userInfo[AVAudioSessionRouteChangeReasonKey] as? UInt,
        let reason = AVAudioSessionRouteChangeReason(rawValue:reasonValue) else {
            return
    }
    switch reason {
    case .newDeviceAvailable:
        let session = AVAudioSession.sharedInstance()
        for output in session.currentRoute.outputs where output.portType ==
AVAudioSessionPortHeadphones {
            headphonesConnected = true
        }
    case .oldDeviceUnavailable:
        if let previousRoute =
            userInfo[AVAudioSessionRouteChangePreviousRouteKey] as?
AVAudioSessionRouteDescription {
            for output in previousRoute.outputs where output.portType ==
AVAudioSessionPortHeadphones {
```

```
                headphonesConnected = false
            }
        }
    default: ()
    }
}
```

Copyright © 2016 Apple Inc. All Rights Reserved. Terms of Use | Privacy Policy | Updated: 2016-10-27