Exploring AVFoundation

The example projects in Building a Basic Playback App showed how easy it is for you to create playback apps using AVKit. For basic video playback, the examples in that chapter may be all you need, but to take advantage of all of the features provided by AVKit, you should have an understanding of the AVFoundation framework objects that drive playback. This chapter explores the essentials of AVFoundation and gives you the information you need to build full-featured video playback apps with AVKit and AVFoundation.

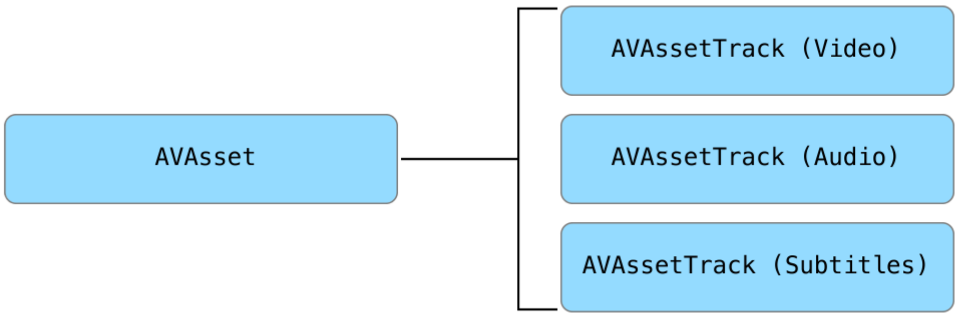
Understanding the Asset Model

Many of AVFoundation's key features and capabilities relate to playing and processing media *assets*. The framework models assets using the `AVAsset` class, which is an abstract, immutable type representing a single media resource. It provides a composite view of a media asset, modeling the static aspects of the media as a whole. An instance of `AVAsset` can model local file-based media, such as a QuickTime movie or an MP3 audio file, but can also represent an asset progressively downloaded from a remote host or streamed using HTTP Live Streaming (HLS).

`AVAsset` simplifies working with media in two important ways. First, it provides a level of independence from the media *format*. It gives you a consistent interface for managing and interacting with your media regardless of its underlying type. The details of working with container formats and codec types are left to the framework, leaving you to focus on how you want to use those assets in your app. Second, `AVAsset` provides a level of independence from the media's *location*. You create an asset instance by initializing it with the media's URL. This could be a local URL, such as one contained within your app bundle or elsewhere on the file system, or it could also be a resource such as an HLS stream hosted on a remote server. In either case, the framework performs the work necessary to efficiently retrieve and load the media in a timely manner on your behalf. Removing the burdens of dealing with media formats and locations greatly simplifies working with audiovisual media.

`AVAsset` is a container object composed of one or more instances of `AVAssetTrack`, which models the asset's uniformly typed media streams. The most commonly used track types are audio and video tracks, but `AVAssetTrack` also models other supplementary tracks such as closed captions, subtitles, and timed metadata (see Figure 3-1).

Figure 3-1 Asset Composition



You retrieve an asset's collection of tracks using its `tracks` property. In many cases, you'll want to perform operations on a subset of an asset's tracks rather than on its complete collection. In these cases, `AVAsset` also provides methods to retrieve subsets of tracks based on criteria such as identifier, media type, or characteristic.

Creating an Asset

You create an AVAsset by initializing it with a local or remote URL pointing to a media resource, as shown in the following example:

```
let url: URL = // Local or Remote Asset URL
let asset = AVAsset(url: url)
```

AVAsset is an abstract class, so when you create an asset as shown in the example, you're actually creating an instance of one of its concrete subclasses called AVURLAsset. In many cases this is a suitable way of creating an asset, but you can also directly instantiate an AVURLAsset when you need more fine-grained control over its initialization. The initializer for AVURLAsset accepts an options dictionary, which lets you tailor the asset's initialization to your particular use case. For instance, if you're creating an asset for an HLS stream, you may want to prevent it from retrieving its media when a user is connected to a cellular network. You could do this as shown in the following example:

```
let url: URL = // Remote Asset URL
let options = [AVURLAssetAllowsCellularAccessKey: false]
let asset = AVURLAsset(url: url, options: options)
```

Passing a value of false for the AVURLAssetAllowsCellularAccessKey option indicates that you want this asset to retrieve its media only when a user is connected to a Wi-Fi network. See *AVURLAsset Class Reference* for information about its available initialization options.

Preparing Assets for Use

You use the properties of AVAsset to determine its features and capabilities, such as its suitability for playback, duration, creation date, and metadata. Creating an asset does not automatically load its properties or prepare it for any particular use. Instead, the loading of an asset's property values is deferred until they are requested. Because property access is *synchronous*, if the requested property hasn't previously been loaded, the framework may need to perform a significant amount of work to return a value. In macOS, this can result in an unresponsive user interface if an unloaded property is accessed from the main thread. In iOS and tvOS, the situation can be even more serious because media operations are performed by the shared media services daemon. If the request to retrieve an unloaded property value is blocked for too long, a timeout occurs resulting in a termination of media services. To prevent this from happening, load an asset's properties *asynchronously*.

AVAsset and AVAssetTrack adopt the AVAsynchronousKeyValueLoading protocol, which defines the methods you use to query the current loaded state of a property and asynchronously load one or more property values, if needed. The protocol defines two methods:

```
public func loadValuesAsynchronously(forKeys keys: [String], completionHandler
handler: (() -> Void)?)
public func statusOfValue(forKey key: String, error outError: NSErrorPointer) ->
AVKeyValueStatus
```

You use the loadValuesAsynchronouslyForKeys:completionHandler: method to asynchronously load one or more property values. You pass it an array of *keys*, which are the names of the properties to load, and a completion block that is called after a status is determined. The following example shows how to asynchronously load an asset's playable property:

```
// URL of a bundle asset called 'example.mp4'
let url = Bundle.main.url(forResource: "example", withExtension: "mp4")!
let asset = AVAsset(url: url)
let playableKey = "playable"

// Load the "playable" property
asset.loadValuesAsynchronously(forKeys: [playableKey]) {
    var error: NSError? = nil
    let status = asset.statusOfValue(forKey: playableKey, error: &error)
    switch status {
    case .loaded:
        // Sucessfully loaded. Continue processing.
    case .failed:
        // Handle error
    case .cancelled:
        // Terminate processing
    default:
        // Handle all other cases
    }
}
```

You check the property's status in the completion callback using the `statusOfValueForKey:error:` method. A status of `AVKeyValueStatusLoaded` indicates the property value was successfully loaded and can be retrieved without blocking. A status of `AVKeyValueStatusFailed` indicates the property value is not available due to an error encountered while attempting to load its data. You can determine the reason for the error by inspecting the `NSError` pointer. In all cases, be aware that the completion callback is invoked on an arbitrary background queue. Dispatch method invocations back to the main queue before performing any user interface-related operations.

Working with Metadata

Media container formats can store descriptive metadata about their media. As a developer, it's often challenging to work with metadata, because each *container format* has its own unique *metadata format*. You typically need a low-level understanding of the format to read and write a container's metadata, but AVFoundation simplifies working with metadata through the use of its `AVMetadataItem` class.

In its most basic form, an instance of `AVMetadataItem` is a key-value pair representing a single metadata value such as a movie's title or an album's artwork. In the same way that `AVAsset` provides a normalized view of your media, `AVMetadataItem` provides a normalized view of its associated metadata.

Retrieving a Collection of Metadata

To effectively use `AVMetadataItem`, you should understand how AVFoundation organizes metadata. To simplify finding and filtering metadata items, the framework groups related metadata into *key spaces*:

- **Format-specific key spaces.** The framework defines a number of format-specific key spaces. These roughly correlate to a particular container or file format, such as QuickTime (Quicktime Metadata and User Data) or MP3 (ID3). However, a single asset may contain metadata values across multiple key spaces. You can retrieve an asset's complete collection of format-specific metadata using its `metadata` property.
- **Common key space.** There are a number of common metadata values, such as a movie's creation date or description, that can exist across multiple key spaces. To help normalize access to this common metadata, the framework provides a *common* key space that gives access to a *limited* set of metadata values common to several key spaces. This makes it easy for you to retrieve commonly used metadata without concern for the specific format. You can retrieve an asset's collection of common metadata using its `commonMetadata` property.

You determine what metadata formats an asset contains by calling its `availableMetadataFormats` property. This property returns an array of string identifiers for each metadata format it contains. You then use its `metadataForFormat:` method to retrieve format-specific metadata values by passing it an appropriate format identifier, as shown below:

```
let url = Bundle.main.url(forResource: "audio", withExtension: "m4a")!
let asset = AVAsset(url: url)
let formatsKey = "availableMetadataFormats"
asset.loadValuesAsynchronously(forKeys: [formatsKey]) {
    var error: NSError? = nil
    let status = asset.statusOfValue(forKey: formatsKey, error: &error)
    if status == .loaded {
        for format in asset.availableMetadataFormats {
            let metadata = asset.metadata(forFormat: format)
            // process format-specific metadata collection
        }
    }
}
```

Finding and Using Metadata Values

After you've retrieved a collection of metadata, the next step is to find the specific values of interest within it. You use the various class methods of `AVMetadataItem` to filter metadata collections down to a discrete set of values. The easiest way to find specific metadata items is to filter by `identifier`, which groups the notion of a key space and key into a single unit. The following example shows how to retrieve the *title* item from the common key space:

```
let metadata = asset.commonMetadata
let titleID = AVMetadataCommonIdentifierTitle
let titleItems = AVMetadataItem.metadataItems(from: metadata, filteredByIdentifier:
titleID)
if let item = titleItems.first {
    // process title item
}
```

Note: The filtering methods of `AVMetadataItem` return collections of items instead of a single instance. In many cases, the returned collection contains a single element, but if the media contains localized metadata or if you are retrieving data from the common keyspace and the same value exists in multiple key spaces, a distinct value matching each locale or keyspace is returned.

After you've retrieved a specific metadata item, the next step is to call its `value` property. The value returned is an object type adopting the `NSObject` and `NSCopying` protocols. You can manually cast the value to the appropriate type, but it's safer and easier to use the metadata item's type coercion properties. You can use its `stringValue`, `numberValue`, `dateValue`, and `dataValue` properties to easily coerce the value to the appropriate type. For instance, the following example shows how you retrieve the artwork associated with an iTunes music track:

```
// Collection of "common" metadata
let metadata = asset.commonMetadata
// Filter metadata to find the asset's artwork
let artworkItems =
    AVMetadataItem.metadataItems(from: metadata,
                                      filteredByIdentifier:
AVMetadataCommonIdentifierArtwork)
```

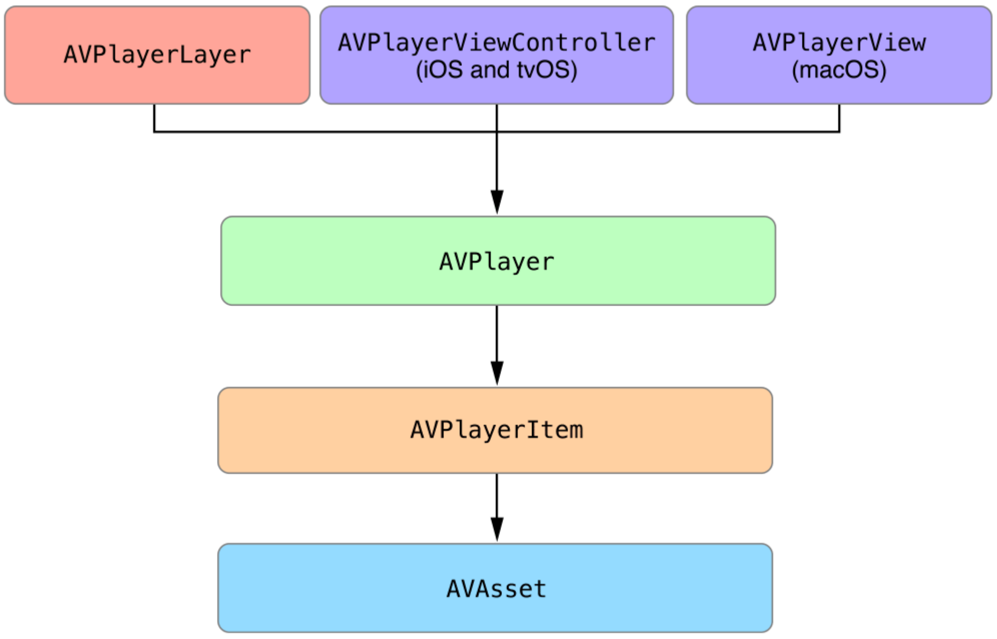
```
if let artworkItem = artworkItems.first {
    // Coerce the value to an NSData using its dataValue property
    if let imageData = artworkItem.dataValue {
        let image = UIImage(data: imageData)
        // process image
    } else {
        // No image data found
    }
}
```

Metadata plays an important role in many media apps. Later sections of this guide explain how you can enhance the capabilities of your playback apps using static and timed metadata.

Playing Media

The asset model described in the previous section is the cornerstone of the playback use case. Assets represent the media you want to play, but are only part of the picture. This section discusses the additional objects needed to play your media and shows how to configure them for playback (see Figure 3-2).

Figure 3-2 Primary Playback Objects



AVPlayer

`AVPlayer` is the central class driving the playback use case. A player is a controller object that manages the playback and timing of a media asset. You use it to play local, progressively downloaded, or streamed media and programmatically control its presentation.

> **Note:** You use `AVPlayer` to play a single media asset at a time. The framework also provides a subclass of `AVPlayer`, called `AVQueuePlayer`, which you use to create and manage a queue of media assets to be played sequentially.

AVPlayerItem

`AVAsset` models only the *static* aspects of the media, such as its duration or creation date, and on its own is unsuitable for playback with `AVPlayer`. To play an asset, you create an instance of its *dynamic* counterpart, found in `AVPlayerItem`. This object models the timing and presentation state of an asset played by `AVPlayer`. Using the properties and methods of `AVPlayerItem`, you can seek to various times in the media, determine its presentation size, identify its current time, and much more.

AVKit and AVPlayerLayer

`AVPlayer` and `AVPlayerItem` are nonvisual objects and on their own are unable to present an asset's video onscreen. You have two different options available to you to display your video content in your app:

- **AVKit.** The best way to present your video content is by using the AVKit framework's `AVPlayerViewController` in iOS or tvOS, or `AVPlayerView` in macOS. These objects present the video content, along with playback controls and other media features, giving you a full-featured playback experience.
- **AVPlayerLayer.** If you're building a custom interface for your player, you'll use a `CALayer` subclass provided by AVFoundation called `AVPlayerLayer`. The player layer can be set as a view's backing layer or can be added directly to the layer hierarchy. Unlike `AVPlayerView` or `AVPlayerViewController`, `AVPlayerLayer` doesn't present any playback controls but simply presents the visual content of a player onscreen. It's up to you to build the playback transport controls to play, pause, and seek through the media.

Setting Up the Playback Objects

The following example shows the steps you take to create the complete object graph for a playback scenario. The example is written for iOS and tvOS, but the same basic steps apply to macOS as well.

```
class PlayerViewController: UIViewController {

    @IBOutlet weak var playerViewController: AVPlayerViewController!

    var player: AVPlayer!
    var playerItem: AVPlayerItem!

    override func viewDidLoad() {
        super.viewDidLoad()

        // 1) Define asset URL
        let url: URL = // URL to local or streamed media

        // 2) Create asset instance
        let asset = AVAsset(url: url)

        // 3) Create player item
        playerItem = AVPlayerItem(asset: asset)

        // 4) Create player instance
        player = AVPlayer(playerItem: playerItem)

        // 5) Associate player with view controller
        playerViewController.player = player
    }

}
```

With the playback objects created, you call the player's `play` method to begin playback.

`AVPlayer` and `AVPlayerItem` provide a variety of ways to control playback when the player item's media is ready for use. The next step is to look at how to observe the state of the playback objects so you can determine their playback readiness.

Observing Playback State

`AVPlayer` and `AVPlayerItem` are dynamic objects whose state changes frequently. You often want to take actions in response to these changes, and the way you do so is by using Key-Value Observing (KVO) (see *Key-Value Observing Programming Guide*). With KVO, an object can register to observe another object's state. When the observed object's state changes, the observer is notified with details of the state change. Using KVO makes it easy for you to observe state changes to `AVPlayer` and `AVPlayerItem` and take actions in response.

One of the most important `AVPlayerItem` properties to observe is its `status`. The `status` indicates if the player item is ready for playback and generally available for use. When you first create a player item, its `status` has a value of `AVPlayerItemStatusUnknown`, meaning its media hasn't been loaded or been enqueued for playback. When you associate a player item with `AVPlayer`, it immediately begins enqueuing the item's media and preparing it for playback. The player item becomes ready for use when its status changes to `AVPlayerItemStatusReadyToPlay`. The following example shows how you can observe this state change:

```
let url: URL = // Asset URL

var asset: AVAsset!
var player: AVPlayer!
var playerItem: AVPlayerItem!

// Key-value observing context
private var playerItemContext = 0

let requiredAssetKeys = [
    "playable",
    "hasProtectedContent"
]

func prepareToPlay() {
    // Create the asset to play
    asset = AVAsset(url: url)

    // Create a new AVPlayerItem with the asset and an
    // array of asset keys to be automatically loaded
    playerItem = AVPlayerItem(asset: asset,
                              automaticallyLoadedAssetKeys: requiredAssetKeys)

    // Register as an observer of the player item's status property
    playerItem.addObserver(self,
                           forKeyPath: #keyPath(AVPlayerItem.status),
                           options: [.old, .new],
                           context: &playerItemContext)

    // Associate the player item with the player
    player = AVPlayer(playerItem: playerItem)
}
```

The `prepareToPlay` method registers to observe the player item's `status` property using the `addObserver:forKeyPath:options:context:` method. Call this method before associating the player item with the player to make sure you capture all state changes to the item's `status`.

To be notified of `status` changes, you implement the `observeValueForKeyPath:ofObject:change:context:` method. This method is invoked whenever the `status` changes, giving you the chance to take some action:

```
override func observeValue(forKeyPath keyPath: String?,
                           of object: Any?,
                           change: [NSKeyValueChangeKey : Any]?,
                           context: UnsafeMutableRawPointer?) {

    // Only handle observations for the playerItemContext
    guard context == &playerItemContext else {
        super.observeValue(forKeyPath: keyPath,
                           of: object,
                           change: change,
                           context: context)
        return
    }

    if keyPath == #keyPath(AVPlayerItem.status) {
        let status: AVPlayerItemStatus
        if let statusNumber = change?[.newKey] as? NSNumber {
            status = AVPlayerItemStatus(rawValue: statusNumber.intValue)!
        } else {
            status = .unknown
        }
        // Switch over status value
        switch status {
        case .readyToPlay:
            // Player item is ready to play.
        case .failed:
            // Player item failed. See error.
        case .unknown:
            // Player item is not yet ready.
        }
    }
}
```

The example retrieves the new `status` value from the change dictionary and switches over its value. If the player item's `status` is `AVPlayerItemStatusReadyToPlay`, then it's ready for use. If a problem is encountered while attempting to load the player item's media, the `status` is `AVPlayerItemStatusFailed`. If a failure occurred, you can retrieve the `NSError` object providing the details of the failure by querying the player item's `error` property.

Performing Time-Based Operations

Media playback is a time-based activity—you present timed media samples at a fixed rate over a certain duration. Time-based operations, such as seeking through the media, play a central role when building media playback apps. Many of the key features of `AVPlayer` and `AVPlayerItem` are related to controlling media timing. To learn use these features effectively, you should understand how time is represented in AVFoundation.

Several Apple frameworks, including some parts of AVFoundation, represent time as a floating-point `NSTimeInterval` value that represents seconds. In many cases, this provides a natural way of thinking about and representing time, but it's often problematic when performing timed media operations. It's important to maintain sample-accurate timing when working with media, and floating-point imprecisions can often result in timing drift. To resolve these imprecisions, AVFoundation represents time using the Core Media framework's `CMTime` data type.

Core Media is a low-level C framework that provides much of the functionality found in AVFoundation and higher-level media frameworks on Apple platforms. In most cases you'll work with Core Media through the higher-level interfaces provided by AVFoundation, but a commonly used data type it provides is called `CMTime`.

```
public struct CMTime {
    public var value: CMTimeValue
    public var timescale: CMTimeScale
    public var flags: CMTimeFlags
    public var epoch: CMTimeEpoch
}
```

This struct defines a rational—or fractional—representation of time. The two most important fields defined by `CMTime` are its value and timescale. `CMTimeValue` is a 64-bit integer defining the numerator of the fractional time, and `CMTimeScale` is a 32-bit integer defining the denominator. This struct makes it easy to represent times expressed in terms of the media's frame rate or sample rate.

```
// 0.25 seconds
let quarterSecond = CMTime(value: 1, timescale: 4)

// 10 second mark in a 44.1 kHz audio file
let tenSeconds = CMTime(value: 441000, timescale: 44100)

// 3 seconds into a 30fps video
let cursor = CMTime(value: 90, timescale: 30)
```

Core Media provides a number of ways to create `CMTime` values and perform arithmetic, comparison, validation, and conversion operations on them. If you're using Swift, Core Media also adds a number of extensions and operator overloads to `CMTime`, making it easy and natural to perform many common operations. See *Core Media Framework Reference* for more information.

Observing Time

You'll commonly want to observe the playback time as it progresses so you can update the playback position or otherwise synchronize the state of your user interface. Earlier you saw how you can use KVO to observe the state of playback objects. KVO works well for general state observations, but isn't the right choice for observing player timing because it's not well suited for observing continuous state changes. Instead, `AVPlayer` provides two different ways for you to observe player time changes: periodic observations and boundary observations.

Periodic Observations

You can observe time ticking by at some regular, periodic interval. If you're building a custom player, the most common use case for periodic observation is to update the time display in your user interface.

To observe periodic timing, you use the player's `addPeriodicTimeObserverForInterval:queue:usingBlock:` method. This method takes a `CMTime` representing the time interval, a *serial* dispatch queue, and a callback block to be invoked at the specified time interval. The following example shows how to set up a block to be called every half-second during normal playback:

```
var player: AVPlayer!
var playerItem: AVPlayerItem!
var timeObserverToken: Any?

func addPeriodicTimeObserver() {
    // Notify every half second
    let timeScale = CMTimeScale(NSEC_PER_SEC)
    let time = CMTime(seconds: 0.5, preferredTimescale: timeScale)
    timeObserverToken = player.addPeriodicTimeObserver(forInterval: time,
                                                       queue: .main) {
        [weak self] time in
        // update player transport UI
    }
}

func removePeriodicTimeObserver() {
    if let timeObserverToken = timeObserverToken {
        player.removeTimeObserver(timeObserverToken)
        self.timeObserverToken = nil
    }
}
```

Boundary Observations

The other way you can observe time is by *boundary*. You can define various points of interest within the media's timeline, and the framework will call you back as those times are traversed during normal playback. Boundary observations are used less frequently than periodic observations, but can still prove useful in certain situations. For instance, you might use boundary observations if you are presenting a video with no playback controls and want to synchronize elements of the display or present supplemental content as those times are traversed.

To observe boundary times, you use the player's `addBoundaryTimeObserverForTimes:queue:usingBlock:` method. This method takes an array of `NSValue` objects wrapping the `CMTime` values that define your boundary times, a *serial* dispatch queue, and a callback block. The following example shows how to define boundary times for each quarter of playback:

```
var asset: AVAsset!
var player: AVPlayer!
var playerItem: AVPlayerItem!
var timeObserverToken: Any?

func addBoundaryTimeObserver() {

    // Divide the asset's duration into quarters.
    let interval = CMTimeMultiplyByFloat64(asset.duration, 0.25)
    var currentTime = kCMTimeZero
    var times = [NSValue]()

    // Calculate boundary times
    while currentTime < asset.duration {
        currentTime = currentTime + interval
        times.append(NSValue(time:currentTime))
    }

    timeObserverToken = player.addBoundaryTimeObserver(forTimes: times,
                                                       queue: .main) {
        // Update UI
    }
}

func removeBoundaryTimeObserver() {
    if let timeObserverToken = timeObserverToken {
        player.removeTimeObserver(timeObserverToken)
        self.timeObserverToken = nil
    }
}
```

Seeking Media

In addition to normal, linear playback, users also want the ability to seek or scrub in a nonlinear manner to quickly get to various points of interest within the media. AVKit automatically provides a scrubbing control for you (if supported by the media), but if you're building a custom player, you'll need to build this feature yourself. Even in cases where you're using AVKit, you still may want to provide a supplemental user interface, such as a table view or a collection view, that lets users quickly skip to various locations in the media.

You can seek in a number of ways using the methods of `AVPlayer` and `AVPlayerItem`. The most common way is to use the player's `seekToTime:` method, passing it a destination `CMTime` value as follows:

```
// Seek to the 2 minute mark
let time = CMTime(value: 120, timescale: 1)
player.seek(to: time)
```

The `seekToTime:` method is a convenient way to quickly seek through your presentation, but it's tuned more for speed rather than precision. This means the actual time to which the player moves may differ from the time you requested. If you need to implement precise seeking behavior, use the `seekToTime:toleranceBefore:toleranceAfter:` method, which lets you indicate the tolerated amount of deviation from your target time (before and after). If you need to provide sample-accurate seeking behavior, you can indicate that zero tolerance is allowed:

```
// Seek to the first frame at 3:25 mark
let seekTime = CMTime(seconds: 205, preferredTimescale: Int32(NSEC_PER_SEC))
player.seek(to: seekTime, toleranceBefore: kCMTimeZero, toleranceAfter: kCMTimeZero)
```

Important: Calling the `seekToTime:toleranceBefore:toleranceAfter:` method with small or zero-valued tolerances may incur additional decoding delay, which can impact your app's seeking behavior.

With a good understanding of how to represent and use time, observe player timing, and seek through media, it's time to look into some more platform-specific features provided by AVKit.