Class

SpeechAnalyzer

Analyzes spoken audio content in various ways and manages the analysis session.

iOS 26.0+ | iPadOS 26.0+ | Mac Catalyst 26.0+ | macOS 26.0+ | visionOS 26.0+

```
final actor SpeechAnalyzer
```

Overview

The Speech framework provides several modules that can be added to an analyzer to provide specific types of analysis and transcription. Many use cases only need a `SpeechTranscriber` module, which performs speech-to-text transcriptions.

The `SpeechAnalyzer` class is responsible for:

- Holding associated modules
- Accepting audio speech input
- Controlling the overall analysis

Each module is responsible for:

- Providing guidance on acceptable input
- Providing its analysis or transcription output

Analysis is asynchronous. Input, output, and session control are decoupled and typically occur over several different tasks created by you or by the session. In particular, where an Objective-C API might use a delegate to provide results to you, the Swift API's modules provides their results via an `AsyncSequence`. Similarly, you provide speech input to this API via an `AsyncSequence` you create and populate.

The analyzer can only analyze one input sequence at a time.

Perform analysis

To perform analysis on audio files and streams, follow these general steps:

1. Create and configure the necessary modules.

2. Ensure the relevant assets are installed or already present. See `AssetInventory`.

3. Create an input sequence you can use to provide the spoken audio.

4. Create and configure the analyzer with the modules and input sequence.

5. Supply audio.

6. Start analysis.

7. Act on results.

8. Finish analysis when desired.

This example shows how you could perform an analysis that transcribes audio using the `Speech Transcriber` module:

```
import Speech

// Step 1: Modules
guard let locale = SpeechTranscriber.supportedLocale(equivalentTo: Locale.current) e
    /* Note unsupported language */
}
let transcriber = SpeechTranscriber(locale: locale, preset: .offlineTranscription)

// Step 2: Assets
if let installationRequest = try await AssetInventory.assetInstallationRequest(suppo
    try await installationRequest.downloadAndInstall()
}

// Step 3: Input sequence
let (inputSequence, inputBuilder) = AsyncStream.makeStream(of: AnalyzerInput.self)

// Step 4: Analyzer
let audioFormat = await SpeechAnalyzer.bestAvailableAudioFormat(compatibleWith: [tra
let analyzer = SpeechAnalyzer(modules: [transcriber])

// Step 5: Supply audio
```

```
Task {
    while /* audio remains */ {
        /* Get some audio */
        /* Convert to audioFormat */
        let pcmBuffer = /* an AVAudioPCMBuffer containing some converted audio */
        let input = AnalyzerInput(buffer: pcmBuffer)
        inputBuilder.yield(input)
    }
    inputBuilder.finish()
}

// Step 7: Act on results
Task {
    do {
        for try await result in transcriber.results {
            let bestTranscription = result.text // an AttributedString
            let plainTextBestTranscription = String(bestTranscription.characters) //
            print(plainTextBestTranscription)
        }
    } catch {
        /* Handle error */
    }
}

// Step 6: Perform analysis
let lastSampleTime = try await analyzer.analyzeSequence(inputSequence)

// Step 8: Finish analysis
if let lastSampleTime {
    try await analyzer.finalizeAndFinish(through: lastSampleTime)
} else {
    try analyzer.cancelAndFinishNow()
}
```

Analyze audio files

To analyze one or more audio files represented by an `AVAudioFile` object, call methods such as `analyzeSequence(from:)` or `start(inputAudioFile:finishAfterFile:)`, or create the analyzer with one of the initializers that has a file parameter. These methods automatically convert the file to a supported audio format and process the file in its entirety.

To end the analysis session after one file, pass `true` for the `finishAfterFile` parameter or call one of the `finish` methods.

Otherwise, by default, the analyzer won't terminate its result streams and will wait for additional audio files or buffers. The analysis session doesn't reset the audio timeline after each file; the next audio is assumed to come immediately after the completed file.

Analyze audio buffers

To analyze audio buffers directly, convert them to a supported audio format, either on the fly or in advance. You can use `bestAvailableAudioFormat(compatibleWith:)` or individual modules' `availableCompatibleAudioFormats` methods to select a format to convert to.

Create an `AnalyzerInput` object for each audio buffer and add the object to an input sequence you create. Supply that input sequence to `analyzeSequence(_:)`, `start(inputSequence:)`, or a similar parameter of the analyzer's initializer.

To skip past part of an audio stream, omit the buffers you want to skip from the input sequence. When you resume analysis with a later buffer, you can ensure the time-code of each module's result accounts for the skipped audio. To do this, pass the later buffer's time-code within the audio stream as the `bufferStartTime` parameter of the later `AnalyzerInput` object.

Analyze autonomously

You can and usually should perform analysis using the `analyzeSequence(_:)` or `analyzeSequence(from:)` methods; those methods work well with Swift structured concurrency techniques. However, you may prefer that the analyzer proceed independently and perform its analysis autonomously as audio input becomes available in a task managed by the analyzer itself.

To use this capability, create the analyzer with one of the initializers that has an input sequence or file parameter, or call `start(inputSequence:)` or `start(inputAudioFile:finishAfterFile:)`. To end the analysis when the input ends, call `finalizeAndFinishThroughEndOfInput()`. To end the analysis of that input and start analysis of different input, call one of the `start` methods again.

Control processing and timing of results

Modules deliver results periodically, but you can manually synchronize their processing and delivery to outside cues.

To deliver a result for a particular time-code, call `finalize(through:)`. To cancel processing of results that are no longer of interest, call `cancelAnalysis(before:)`.

Improve responsiveness

By default, the analyzer and modules load the system resources that they require lazily, and unload those resources when they're deallocated.

To proactively load system resources and "preheat" the analyzer, call `prepareToAnalyze(in:)` after setting its modules. This may improve how quickly the modules return their first results.

To delay or prevent unloading an analyzer's resources—caching them for later use by a different analyzer instance—you can select a `SpeechAnalyzer.Options.ModelRetention` option and create the analyzer with an appropriate `SpeechAnalyzer.Options` object.

To set the priority of analysis work, create the analyzer with a `SpeechAnalyzer.Options` object given a `priority` value.

Specific modules may also offer options that improve responsiveness.

Finish analysis

To end an analysis session, you must use one of the analyzer's `finish` methods or parameters, or deallocate the analyzer.

When the analysis session transitions to the *finished* state:

- The analyzer won't take additional input from the input sequence
- Most methods won't do anything; in particular, the analyzer won't accept different input sequences or modules
- Module result streams terminate and modules won't publish additional results, though the app can continue to iterate over already-published results

> **Note**
>
> While you can terminate the input sequence you created with a method such as `AsyncStream.Continuation.finish()`, finishing the input sequence does *not* cause the analysis session to become finished, and you can continue the session with a different input sequence.

Respond to errors

When the analyzer or its modules' result streams throw an error, the analysis session becomes finished as described above, and the same error (or a `CancellationError`) is thrown from all waiting methods and result streams.

Topics

Creating an analyzer

`convenience init(modules: [any SpeechModule], options: SpeechAnalyzer.Options?)`

Creates an analyzer.

`convenience init<InputSequence>(inputSequence: InputSequence, modules: [any SpeechModule], options: SpeechAnalyzer.Options?, analysisContext: AnalysisContext, volatileRangeChangedHandler: sending ((CMTimeRange, Bool, Bool) -> Void)?)`

Creates an analyzer and begins analysis.

`convenience init(inputAudioFile: AVAudioFile, modules: [any SpeechModule], options: SpeechAnalyzer.Options?, analysisContext: AnalysisContext, finishAfterFile: Bool, volatileRangeChangedHandler: sending ((CMTimeRange, Bool, Bool) -> Void)?) async throws`

Creates an analyzer and begins analysis on an audio file.

`struct Options`

Analysis processing options.

Managing modules

`func setModules([any SpeechModule]) async throws`

Adds or removes modules.

`var modules: [any SpeechModule]`

The modules performing analysis on the audio input.

Performing analysis

`func analyzeSequence<InputSequence>(InputSequence) async throws -> CMTime?`

Analyzes an input sequence, returning when the sequence is consumed.

`func analyzeSequence(from: AVAudioFile) async throws -> CMTime?`

Analyzes an input sequence created from an audio file, returning when the file has been read.

Performing autonomous analysis

```
func start<InputSequence>(inputSequence: InputSequence) async throws
```

Starts analysis of an input sequence and returns immediately.

```
func start(inputAudioFile: AVAudioFile, finishAfterFile: Bool) async throws
```

Starts analysis of an input sequence created from an audio file and returns immediately.

Finalizing and cancelling results

```
func cancelAnalysis(before: CMTime)
```

Stops analyzing audio predating the given time.

```
func finalize(through: CMTime?) async throws
```

Finalizes the modules' analyses.

Finishing analysis

```
func cancelAndFinishNow() async
```

Finishes analysis immediately.

```
func finalizeAndFinishThroughEndOfInput() async throws
```

Finishes analysis after an audio input sequence has been fully consumed and its results are finalized.

```
func finalizeAndFinish(through: CMTime) async throws
```

Finishes analysis after finalizing results for a given time-code.

```
func finish(after: CMTime) async throws
```

Finishes analysis once input for a given time is consumed.

Determining audio formats

```
static func bestAvailableAudioFormat(compatibleWith: [any SpeechModule]) async -> AVAudioFormat?
```

Retrieves the best-quality audio format that the specified modules can work with, from assets installed on the device.

```
static func bestAvailableAudioFormat(compatibleWith: [any SpeechModule
], considering: AVAudioFormat?) async -> AVAudioFormat?
```

Retrieves the best-quality audio format that the specified modules can work with, taking into account the natural format of the audio and assets installed on the device.

Improving responsiveness

```
func prepareToAnalyze(in: AVAudioFormat?) async throws
```

Prepares the analyzer to begin work with minimal startup delay.

```
func prepareToAnalyze(in: AVAudioFormat?, withProgressReadyHandler:
sending ((Progress) -> Void)?) async throws
```

Prepares the analyzer to begin work with minimal startup delay, reporting the progress of that preparation.

Monitoring analysis

```
func setVolatileRangeChangedHandler(sending ((CMTimeRange, Bool, Bool)
-> Void)?)
```

A closure that the analyzer calls when the volatile range changes.

```
var volatileRange: CMTimeRange?
```

The range of results that can change.

Managing contexts

```
func setContext(AnalysisContext) async throws
```

Sets contextual information to improve or inform the analysis.

```
var context: AnalysisContext
```

An object containing contextual information.

Relationships

Conforms To

`Actor`, `Sendable`, `SendableMetatype`

See Also

Essentials

{} Bringing advanced speech-to-text capabilities to your app

Learn how to incorporate live speech-to-text transcription into your app with SpeechAnalyzer.

`class AssetInventory`

Manages the assets that are necessary for transcription or other analyses.