**AudioScribe Audio System Design: Handling Route Changes & Interruptions**

**Audio Environment**

AudioScribe uses:

* AVAudioSession to configure iOS audio behavior.
* AVAudioEngine to record microphone input via an installed tap on the input node.

The session is set up in AudioManager.configureSession():

try audioSession.setCategory(.playAndRecord, mode: .default, options: [.defaultToSpeaker, .allowBluetooth])

### try audioSession.setActive(true, options: .notifyOthersOnDeactivation) **Why this matters:**

* .playAndRecord ensures we can both record and play audio (e.g., play waveforms back).
* .defaultToSpeaker automatically routes playback to the speaker instead of the receiver.
* .allowBluetooth lets users use Bluetooth microphones or headsets.
* .notifyOthersOnDeactivation allows AudioScribe to gracefully yield audio focus to other apps, avoiding clashes.

**Audio Route Change Handling**

Audio route changes happen when:

* Headphones or a Bluetooth mic are plugged/unplugged.
* Switching from speaker to receiver.
* Switching between audio devices.

**The current implementation:**

* Registers for notifications:

NotificationCenter.default.addObserver(

self,

selector: #selector(handleRouteChange),

name: AVAudioSession.routeChangeNotification,

object: nil

* )  
    
  Inside handleRouteChange, it inspects the AVAudioSessionRouteChangeReason:

switch reason {

case .oldDeviceUnavailable:

logger.info("Old device removed.")

if isRecording {

restartAudioEngine()

}

...

}

**Restarting the engine** after unplugging ensures the microphone input correctly reattaches, preventing “silent” input bugs that happen when audio taps lose their route.

## Interruption Handling (Phone calls, Siri, etc.)

Interruptions happen when:

* A phone call or FaceTime comes in.
* Siri activates.
* Another app requests exclusive audio.

### The current implementation:

* Registers:

NotificationCenter.default.addObserver(

self,

selector: #selector(handleInterruption),

name: AVAudioSession.interruptionNotification,

object: nil

)

* Inside handleInterruption:

if type == .began {

if isRecording {

stopRecording()

}

} else if type == .ended {

if options.contains(.shouldResume) && !isRecording {

startRecording()

}

}

**This is best practice**:

* On .began, we immediately stop recording. This ensures buffers are flushed, and AVAudioEngine is gracefully stopped.
* On .ended, we check if iOS suggests resuming (.shouldResume) and restart the recording session if appropriate.

## Why This Design?

* **Resilient**: guarantees audio recording doesn’t hang if a mic is unplugged.
* **Battery-friendly**: stopping the engine during interruptions lets iOS deal with hardware.
* **Follows Apple guidlines**: recommended by AVAudioSession best practices for handling live audio capture.

## Possible Future Improvements

I think the design is solid for typical needs, but we could extend it to:

* It might be nice to reinstate the user’s paused state instead of always auto resuming after interruption.
* Show a UI message like “Recording paused because a call came in.”, etc.