**import** SwiftUI

**struct** ContentView: View {

@StateObject **private** **var** viewModel = SpeechViewModel()

**var** body: **some** View {

VStack(spacing: 16) {

// Title

Text("Voice Assistant")

.font(.largeTitle)

.fontWeight(.bold)

.padding(.top)

// Language selection

Picker("Language", selection: $viewModel.selectedLanguage) {

ForEach(viewModel.supportedLanguages, id: \.**self**) { language **in**

Text(language.name).tag(language)

}

}

.pickerStyle(MenuPickerStyle())

.padding(.horizontal)

// Waveform when recording

**if** viewModel.isRecording {

WaveformView(audioLevelMonitor: viewModel.audioService.audioLevelMonitor)

.transition(.opacity)

.animation(.easeInOut, value: viewModel.isRecording)

}

// Text boxes

textBoxes

// Buttons

buttonRow

// Error message

**if** **let** errorMessage = viewModel.errorMessage {

Text(errorMessage)

.foregroundColor(.red)

.font(.footnote)

.padding()

.multilineTextAlignment(.center)

}

}

.padding()

}

**private** **var** textBoxes: **some** View {

VStack(spacing: 16) {

// Original text box

VStack(alignment: .leading) {

Text("Original Text")

.font(.headline)

TextEditor(text: $viewModel.speechText.originalText)

.frame(minHeight: 120)

.padding(8)

.background(Color(.systemGray6))

.cornerRadius(8)

.overlay(

RoundedRectangle(cornerRadius: 8)

.stroke(Color.gray, lineWidth: 1)

)

}

// Processed text box

VStack(alignment: .leading) {

Text("Processed Text")

.font(.headline)

TextEditor(text: $viewModel.speechText.processedText)

.frame(minHeight: 120)

.padding(8)

.background(Color(.systemGray6))

.cornerRadius(8)

.overlay(

RoundedRectangle(cornerRadius: 8)

.stroke(Color.gray, lineWidth: 1)

)

}

}

}

**private** **var** buttonRow: **some** View {

HStack(spacing: 12) {

// Record button

Button(action: viewModel.toggleRecording) {

HStack {

Image(systemName: viewModel.isRecording ? "stop.circle.fill" : "mic.circle.fill")

Text(viewModel.isRecording ? "Stop" : "Record")

}

.frame(maxWidth: .infinity)

}

.buttonStyle(PrimaryButtonStyle(isDestructive: viewModel.isRecording))

.disabled(viewModel.isProcessing)

// Translate button

Button(action: viewModel.translateText) {

HStack {

Image(systemName: "globe")

Text("Translate")

}

.frame(maxWidth: .infinity)

}

.buttonStyle(PrimaryButtonStyle())

.disabled(viewModel.isProcessing || viewModel.speechText.originalText.isEmpty)

// Improve button

Button(action: viewModel.improveText) {

HStack {

Image(systemName: "wand.and.stars")

Text("Improve")

}

.frame(maxWidth: .infinity)

}

.buttonStyle(PrimaryButtonStyle())

.disabled(viewModel.isProcessing || viewModel.speechText.originalText.isEmpty)

}

.overlay(

Group {

**if** viewModel.isProcessing {

HStack {

ProgressView()

.progressViewStyle(CircularProgressViewStyle())

Text("Processing...")

}

.padding(8)

.background(Color(.systemBackground).opacity(0.8))

.cornerRadius(8)

}

}

)

}

}

// Custom button style

**struct** PrimaryButtonStyle: ButtonStyle {

**var** isDestructive: Bool = **false**

**func** makeBody(configuration: Configuration) -> **some** View {

configuration.label

.padding(.vertical, 12)

.background(isDestructive ? Color.red : Color.blue)

.foregroundColor(.white)

.cornerRadius(10)

.scaleEffect(configuration.isPressed ? 0.98 : 1)

.opacity(configuration.isPressed ? 0.9 : 1)

}

}

// Helper extension for multipart form data

**extension** Data {

**mutating** **func** append(\_ string: String) {

**if** **let** data = string.data(using: .utf8) {

**self**.append(data)

}

}

}

**struct** ContentView\_Previews: PreviewProvider {

**static** **var** previews: **some** View {

ContentView()

}

}

**import** SwiftUI

**struct** WaveformView: View {

@ObservedObject **var** audioLevelMonitor: AudioLevelMonitor

**var** body: **some** View {

VStack {

// Timer display

Text(timeString(from: audioLevelMonitor.recordingDuration))

.font(.system(size: 40, weight: .medium, design: .monospaced))

.foregroundColor(.secondary)

.padding(.bottom)

// Waveform visualization

HStack(spacing: 4) {

ForEach(0..<audioLevelMonitor.soundSamples.count, id: \.**self**) { index **in**

BarView(value: audioLevelMonitor.soundSamples[index])

}

}

.frame(height: 100)

.padding()

}

}

**private** **func** timeString(from timeInterval: TimeInterval) -> String {

**let** minutes = Int(timeInterval) / 60

**let** seconds = Int(timeInterval) % 60

**let** hundredths = Int((timeInterval.truncatingRemainder(dividingBy: 1)) \* 100)

**return** String(format: "%02d:%02d.%02d", minutes, seconds, hundredths)

}

}

**struct** BarView: View {

**var** value: CGFloat

**var** body: **some** View {

ZStack {

Rectangle()

.fill(WaveformConfig.waveColor)

.frame(width: 3, height: max(value \* 100, 3))

}

}

}

// **MARK: - Models**

**import** Foundation

**import** AVFoundation

**import** SwiftUI

// API Keys configuration

**struct** APIConfig {

// For a real app, you'd want to store this securely

**static** **var** openAIKey: String {

// First try to get from environment

**if** **let** envKey = ProcessInfo.processInfo.environment["OPENAI\_API\_KEY"] {

**return** envKey

}

// Add your key here for development purposes only

// Don't include this in production code or source control

**return** "xxxx"

}

}

// Configuration for the OpenAI models

**struct** ModelConfig {

**static** **let** transcriptionModel = "gpt-4o-mini-transcribe"

**static** **let** llmModel = "gpt-4o-mini"

}

// Language model for translation

**struct** Language: Identifiable, Hashable {

**let** id = UUID()

**let** name: String

**let** code: String

}

// Models related to speech and text functionality

**struct** SpeechText {

**var** originalText: String = ""

**var** processedText: String = ""

}

// Error model

**enum** AppError: Error {

**case** recordingError(String)

**case** transcriptionError(String)

**case** processingError(String)

**case** networkError(String)

**case** missingAPIKey

**var** description: String {

**switch** **self** {

**case** .recordingError(**let** message): **return** "Recording error: \(message)"

**case** .transcriptionError(**let** message): **return** "Transcription error: \(message)"

**case** .processingError(**let** message): **return** "Processing error: \(message)"

**case** .networkError(**let** message): **return** "Network error: \(message)"

**case** .missingAPIKey: **return** "OpenAI API key is missing. Please set it in your app."

}

}

}

**import** Foundation

// Audio file structure

**struct** AudioFile {

**let** data: Data

**let** filename: String

}

// Audio transcription request parameters

**struct** AudioTranscriptionQuery {

**let** file: AudioFile

**let** model: String

**let** language: String?

}

// Audio transcription result

**struct** AudioTranscriptionResult: Decodable {

**let** text: String

}

// Chat role

**enum** ChatRole: String, Codable {

**case** system

**case** user

**case** assistant

}

// Chat message

**struct** ChatMessage: Codable {

**let** role: ChatRole

**let** content: String

}

// Chat completion request

**struct** ChatCompletionRequest: Codable {

**let** model: String

**let** messages: [ChatMessage]

}

// Chat completion response

**struct** ChatCompletionResponse: Decodable {

**struct** Choice: Decodable {

**struct** Message: Decodable {

**let** role: String

**let** content: String?

}

**let** message: Message

**let** index: Int

}

**let** id: String

**let** object: String

**let** created: Int

**let** model: String

**let** choices: [Choice]

}

**import** Foundation

**import** AVFoundation

**class** AudioLevelMonitor: ObservableObject {

@Published **var** soundSamples: [CGFloat] = Array(repeating: 0, count: WaveformConfig.numberOfSamples)

@Published **var** recordingDuration: TimeInterval = 0

**private** **var** audioRecorder: AVAudioRecorder?

**private** **var** timer: Timer?

**private** **var** startTime: Date?

**func** startMonitoring(audioRecorder: AVAudioRecorder) {

**self**.audioRecorder = audioRecorder

**self**.audioRecorder?.isMeteringEnabled = **true**

**self**.soundSamples = Array(repeating: 0, count: WaveformConfig.numberOfSamples)

**self**.recordingDuration = 0

startTime = Date()

timer = Timer.scheduledTimer(withTimeInterval: WaveformConfig.updateInterval, repeats: **true**) { [**weak** **self**] \_ **in**

**self**?.updateMeters()

}

}

**func** stopMonitoring() {

timer?.invalidate()

timer = **nil**

startTime = **nil**

recordingDuration = 0

}

**private** **func** updateMeters() {

**guard** **let** audioRecorder = audioRecorder **else** { **return** }

audioRecorder.updateMeters()

// Update recording duration

**if** **let** startTime = startTime {

recordingDuration = Date().timeIntervalSince(startTime)

}

// Shift samples and add the new value

soundSamples.removeFirst()

// Get the power in decibels

**let** decibels = audioRecorder.averagePower(forChannel: 0)

// Convert decibels to linear scale (0.0 to 1.0)

// Audio levels in decibels are typically negative, with 0dB being the maximum

// and -160dB being the minimum. We normalize this to 0.0-1.0 range.

**var** level: CGFloat = 0

**if** decibels > -50 { // Only show values above a certain threshold

// Convert from decibels (-160...0) to linear (0...1) with some normalization

level = CGFloat(pow(10, Float(decibels) / 20))

// Apply some scaling for better visualization

level = min(max(level, CGFloat(WaveformConfig.minAmplitude)), CGFloat(WaveformConfig.maxAmplitude))

}

soundSamples.append(level)

}

}

**import** SwiftUI

**import** Foundation

// Waveform Configuration

**struct** WaveformConfig {

**static** **let** numberOfSamples = 30 // Number of bars in the visualization

**static** **let** updateInterval: TimeInterval = 0.05 // Update every 50ms

**static** **let** minAmplitude: Float = 0.05 // Minimum amplitude to show

**static** **let** maxAmplitude: Float = 1.0 // Max amplitude

**static** **let** waveColor = Color.red // Color of the waveform

}

**import** Foundation

// Service for handling OpenAI API communications

**class** OpenAIService {

**private** **let** apiKey: String

**private** **let** baseURL = URL(string: "https://api.openai.com/v1/")!

**init**() {

**self**.apiKey = APIConfig.openAIKey

}

**func** transcribeAudio(fileURL: URL, language: String = "en", completion: **@escaping** (Result<String, AppError>) -> Void) {

**guard** !apiKey.isEmpty **else** {

completion(.failure(.missingAPIKey))

**return**

}

// Convert to Data

**do** {

**let** audioData = **try** Data(contentsOf: fileURL)

**let** boundary = UUID().uuidString

**var** request = URLRequest(url: baseURL.appendingPathComponent("audio/transcriptions"))

request.httpMethod = "POST"

request.addValue("Bearer \(apiKey)", forHTTPHeaderField: "Authorization")

request.setValue("multipart/form-data; boundary=\(boundary)", forHTTPHeaderField: "Content-Type")

// Create multipart form data

**var** body = Data()

// Add file

body.append("--\(boundary)\r\n")

body.append("Content-Disposition: form-data; name=\"file\"; filename=\"recording.m4a\"\r\n")

body.append("Content-Type: audio/m4a\r\n\r\n")

body.append(audioData)

body.append("\r\n")

// Add model

body.append("--\(boundary)\r\n")

body.append("Content-Disposition: form-data; name=\"model\"\r\n\r\n")

body.append(ModelConfig.transcriptionModel)

body.append("\r\n")

// Add language if provided

body.append("--\(boundary)\r\n")

body.append("Content-Disposition: form-data; name=\"language\"\r\n\r\n")

body.append(language)

body.append("\r\n")

// Add response format

body.append("--\(boundary)\r\n")

body.append("Content-Disposition: form-data; name=\"response\_format\"\r\n\r\n")

body.append("text")

body.append("\r\n")

// Close the boundary

body.append("--\(boundary)--\r\n")

request.httpBody = body

**let** task = URLSession.shared.dataTask(with: request) { data, response, error **in**

**if** **let** error = error {

completion(.failure(.networkError(error.localizedDescription)))

**return**

}

**guard** **let** httpResponse = response **as**? HTTPURLResponse **else** {

completion(.failure(.networkError("Invalid response")))

**return**

}

**guard** httpResponse.statusCode == 200, **let** data = data **else** {

**let** errorMessage = data.flatMap { String(data: $0, encoding: .utf8) } ?? "Unknown error"

completion(.failure(.transcriptionError("\(httpResponse.statusCode): \(errorMessage)")))

**return**

}

// For text response format, the response is just plain text

**if** **let** text = String(data: data, encoding: .utf8) {

completion(.success(text))

} **else** {

completion(.failure(.transcriptionError("Could not decode response")))

}

}

task.resume()

} **catch** {

completion(.failure(.recordingError("Failed to read audio file: \(error.localizedDescription)")))

}

}

**func** chatCompletion(messages: [ChatMessage], completion: **@escaping** (Result<String, AppError>) -> Void) {

**guard** !apiKey.isEmpty **else** {

completion(.failure(.missingAPIKey))

**return**

}

**let** url = baseURL.appendingPathComponent("chat/completions")

**var** request = URLRequest(url: url)

request.httpMethod = "POST"

request.addValue("Bearer \(apiKey)", forHTTPHeaderField: "Authorization")

request.addValue("application/json", forHTTPHeaderField: "Content-Type")

**let** chatRequest = ChatCompletionRequest(

model: ModelConfig.llmModel,

messages: messages

)

**do** {

**let** encoder = JSONEncoder()

request.httpBody = **try** encoder.encode(chatRequest)

} **catch** {

completion(.failure(.processingError("Failed to encode request: \(error.localizedDescription)")))

**return**

}

**let** task = URLSession.shared.dataTask(with: request) { data, response, error **in**

**if** **let** error = error {

completion(.failure(.networkError(error.localizedDescription)))

**return**

}

**guard** **let** httpResponse = response **as**? HTTPURLResponse **else** {

completion(.failure(.networkError("Invalid response")))

**return**

}

**guard** httpResponse.statusCode == 200, **let** data = data **else** {

**let** errorMessage = data.flatMap { String(data: $0, encoding: .utf8) } ?? "Unknown error"

completion(.failure(.processingError("\(httpResponse.statusCode): \(errorMessage)")))

**return**

}

**do** {

**let** decoder = JSONDecoder()

**let** result = **try** decoder.decode(ChatCompletionResponse.**self**, from: data)

**if** **let** choice = result.choices.first, **let** content = choice.message.content {

completion(.success(content))

} **else** {

completion(.failure(.processingError("No response content")))

}

} **catch** {

completion(.failure(.processingError("Failed to decode response: \(error.localizedDescription)")))

}

}

task.resume()

}

**func** translateText(text: String, targetLanguage: String, completion: **@escaping** (Result<String, AppError>) -> Void) {

**let** messages = [

ChatMessage(role: .system, content: "You are a helpful translation assistant."),

ChatMessage(role: .user, content: "Translate the following text to \(targetLanguage):\n\n\(text)")

]

chatCompletion(messages: messages, completion: completion)

}

**func** improveText(text: String, completion: **@escaping** (Result<String, AppError>) -> Void) {

**let** messages = [

ChatMessage(role: .system, content: "You are a writing improvement assistant. Please improve the following text by correcting grammar, enhancing clarity, and making it more coherent while maintaining the original meaning."),

ChatMessage(role: .user, content: text)

]

chatCompletion(messages: messages, completion: completion)

}

}

**import** Foundation

**import** AVFoundation

// Service for handling audio recording

**class** AudioService: NSObject, ObservableObject {

**private** **var** audioRecorder: AVAudioRecorder?

**private** **var** recordingSession: AVAudioSession?

**private** **var** audioFilePath: URL?

@Published **var** isRecording = **false**

@Published **var** error: AppError?

// Add audio level monitor for visualization

**let** audioLevelMonitor = AudioLevelMonitor()

**override** **init**() {

**super**.init()

setupAudioSession()

}

**private** **func** setupAudioSession() {

recordingSession = AVAudioSession.sharedInstance()

**do** {

**try** recordingSession?.setCategory(.playAndRecord, mode: .default)

**try** recordingSession?.setActive(**true**)

} **catch** {

**self**.error = .recordingError("Failed to set up recording session: \(error.localizedDescription)")

}

}

**func** startRecording() {

// Check permissions

recordingSession?.requestRecordPermission { [**weak** **self**] allowed **in**

**guard** **let** self = **self** **else** { **return** }

DispatchQueue.main.async {

**if** allowed {

self.initiateRecording()

} **else** {

self.error = .recordingError("Microphone access denied")

}

}

}

}

**private** **func** initiateRecording() {

**let** documentsDirectory = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)[0]

audioFilePath = documentsDirectory.appendingPathComponent("recording.m4a")

**let** settings = [

AVFormatIDKey: Int(kAudioFormatMPEG4AAC),

AVSampleRateKey: 44100,

AVNumberOfChannelsKey: 1,

AVEncoderAudioQualityKey: AVAudioQuality.high.rawValue

]

**do** {

audioRecorder = **try** AVAudioRecorder(url: audioFilePath!, settings: settings)

audioRecorder?.record()

isRecording = **true**

// Start monitoring audio levels for visualization

audioLevelMonitor.startMonitoring(audioRecorder: audioRecorder!)

} **catch** **let** recordingError {

**self**.error = .recordingError("Could not start recording: \(recordingError.localizedDescription)")

}

}

**func** stopRecording() -> URL? {

audioRecorder?.stop()

isRecording = **false**

// Stop monitoring audio levels

audioLevelMonitor.stopMonitoring()

**return** audioFilePath

}

}

**import** Foundation

**class** SpeechViewModel: ObservableObject {

@Published **var** speechText = SpeechText()

@Published **var** isProcessing = **false**

@Published **var** isRecording = **false**

@Published **var** errorMessage: String?

@Published **var** selectedLanguage: Language

**let** supportedLanguages = [

Language(name: "English", code: "en"),

Language(name: "Spanish", code: "es"),

Language(name: "French", code: "fr"),

Language(name: "German", code: "de"),

Language(name: "Chinese", code: "zh"),

Language(name: "Japanese", code: "ja"),

Language(name: "Korean", code: "ko"),

Language(name: "Russian", code: "ru"),

Language(name: "Arabic", code: "ar"),

Language(name: "Hindi", code: "hi")

]

// Make audioService accessible to the view for waveform visualization

**let** audioService = AudioService()

**private** **let** openAIService = OpenAIService()

**init**() {

// Default language is English

selectedLanguage = supportedLanguages[0]

// Bind to audio service recording state

audioService.$isRecording

.assign(to: &$isRecording)

// Handle errors from audio service

audioService.$error

.compactMap { $0?.description }

.assign(to: &$errorMessage)

}

**func** toggleRecording() {

**if** isRecording {

stopRecording()

} **else** {

startRecording()

}

}

**private** **func** startRecording() {

errorMessage = **nil**

audioService.startRecording()

}

**private** **func** stopRecording() {

isProcessing = **true**

**guard** **let** audioFileURL = audioService.stopRecording() **else** {

errorMessage = "Failed to get recording file"

isProcessing = **false**

**return**

}

// Transcribe the audio

openAIService.transcribeAudio(fileURL: audioFileURL) { [**weak** **self**] result **in**

**guard** **let** self = **self** **else** { **return** }

DispatchQueue.main.async {

self.isProcessing = **false**

**switch** result {

**case** .success(**let** transcribedText):

self.speechText.originalText = transcribedText

**case** .failure(**let** error):

self.errorMessage = error.description

}

}

}

}

**func** translateText() {

**guard** !speechText.originalText.isEmpty **else** {

errorMessage = "No text to translate"

**return**

}

isProcessing = **true**

errorMessage = **nil**

openAIService.translateText(

text: speechText.originalText,

targetLanguage: selectedLanguage.name

) { [**weak** **self**] result **in**

**guard** **let** self = **self** **else** { **return** }

DispatchQueue.main.async {

self.isProcessing = **false**

**switch** result {

**case** .success(**let** translatedText):

self.speechText.processedText = translatedText

**case** .failure(**let** error):

self.errorMessage = error.description

}

}

}

}

**func** improveText() {

**guard** !speechText.originalText.isEmpty **else** {

errorMessage = "No text to improve"

**return**

}

isProcessing = **true**

errorMessage = **nil**

openAIService.improveText(text: speechText.originalText) { [**weak** **self**] result **in**

**guard** **let** self = **self** **else** { **return** }

DispatchQueue.main.async {

self.isProcessing = **false**

**switch** result {

**case** .success(**let** improvedText):

self.speechText.processedText = improvedText

**case** .failure(**let** error):

self.errorMessage = error.description

}

}

}

}

}