A Simple Tutorial for Building Privacy-First AI Features with Apple's Foundation Models

A hands-on introduction to on-device AI in iOS 26

Praveen Kumar Addiki
addikipraveenkumar0726@gmail.com

Apple announced the Foundation Models framework at WWDC 2025, giving developers direct access to the same on-device AI model that powers Apple Intelligence — running entirely on the device. No cloud. No API costs. No data leaving the phone.

This tutorial walks through the framework for iOS developers who want to get started quickly with practical, production-ready examples.

The basics are demonstrated by building something practical: a transaction categorisation feature for a banking app.

Why Should You Care?

If you've worked with cloud-based AI services like OpenAI or Claude, you know the drill — API keys, network calls, usage costs, and privacy concerns about sending user data to external servers.

Foundation Models changes that:

Everything runs on-device: The 3 billion parameter model lives on the user's iPhone or Mac.
No internet required: Works completely offline.
No cost per request: Use it as much as you want.
Privacy by default: User data never leaves the device.

For apps dealing with sensitive information — banking, health, personal finance — this is a big deal.

What You'll Need

Before we start, make sure you have:

- Xcode 26
- A device running iOS 26 (or macOS Tahoe)
- Apple Intelligence enabled on the device

The framework works across iOS, iPadOS, macOS, and visionOS.

Step 1: Check If the Model Is Available

Not every device supports Foundation Models. The first thing you should do is check availability:

```
import SwiftUI
import FoundationModels

struct ContentView: View {
    private var model = SystemLanguageModel.default

    var body: some View {
        switch model.availability {
```

```
        case .available:
            Text("✅ Ready to go!")
        case .unavailable(let reason):
            Text("❌ Not available: \(reason)")
        @unknown default:
            Text("Unknown status")
        }
    }
}
```

If you see "Ready to go!", you're set. If not, check that Apple Intelligence is enabled in Settings.

Step 2: Your First AI Request

Let's ask the model to categorise a bank transaction. The simplest approach:

```
import FoundationModels

func categoriseTransaction() async {
    let session = LanguageModelSession()

    do {
        let response = try await session.respond(
            to: "Categorise this bank transaction: £45.99 at Tesco Express"
        )
        print(response.content)
        // Output: "This appears to be a grocery purchase at a
supermarket."
    } catch {
        print("Error: \(error)")
    }
}
```

That's it. Three lines of code to get an AI response.

But here's the problem — you get plain text back. In a real app, you need structured data. How do you extract the category? The confidence level? The merchant type?

This is where @Generable comes in.

Step 3: Getting Structured Data with @Generable

Instead of parsing text, you can tell the model: "Give me a Swift struct."

First, define what you need:

```
import FoundationModels

@Generable
struct TransactionCategory {
    let category: String
    let confidence: String
    let merchantType: String
    let reason: String
}
```

The @Generable macro tells the framework: "I want the AI to fill this struct, not return plain text."

This eliminates fragile string parsing and gives you type-safe, ready-to-use data directly from the model — one of the framework's most practical features for production apps.

Now request it:

```
let session = LanguageModelSession()

let response = try await session.respond(
    to: "Categorise this bank transaction: £45.99 at Tesco Express",
    generating: TransactionCategory.self
)

// response.content is now a TransactionCategory struct!
print(response.content.category)      // "Groceries"
print(response.content.confidence)    // "High"
print(response.content.merchantType)  // "Supermarket"
print(response.content.reason)        // "Tesco is a UK supermarket chain"
```

No JSON parsing. No string manipulation. The model directly populates your Swift struct.

Important: You Define the Structure, AI Fills the Values

This is a key concept to understand. When you create a @Generable struct:

You decide what properties you want (category, confidence, merchantType, etc.)
The AI decides what values to put in those properties

Think of it like a form. You design the form with empty fields. The AI fills in the answers based on its understanding.

For example, when the AI sees "Tesco Express":
1. It knows Tesco is a UK supermarket chain
2. It determines this is a grocery purchase
3. It feels confident about this assessment
4. So it fills: `category: "Groceries"`, `confidence: "High"`

But if the AI sees an unknown merchant like "J Smith Services":
1. It doesn't recognise the name
2. It has to guess what category it might be
3. It's uncertain about its assessment
4. So it might fill: `category: "Other"`, `confidence: "Low"`

The AI makes these judgement calls based on its training knowledge — just like a human would.

This is powerful for building UIs because you get ready-to-use data:

```
struct TransactionRow: View {
    let result: TransactionCategory

    var body: some View {
        HStack {
            Text(result.category)
            Spacer()
            Text(result.confidence)
                .foregroundColor(result.confidence == "High" ? .green :
.orange)
        }
    }
```

```
}
```

Step 4: Guiding the Model with @Guide

There's one issue with the basic @Generable approach. The AI might return inconsistent values:

- Sometimes "Groceries", sometimes "Food", sometimes "grocery"
- Sometimes "High", sometimes "Very Confident", sometimes "95%"

The AI is still filling in correct answers, but the format varies. This breaks your UI code:

```
// This might fail because AI returned "high" instead of "High"
switch result.confidence {
case "High": return Color.green
case "Medium": return Color.orange
case "Low": return Color.red
default: return Color.gray  // Unexpected values end up here
}
```

To fix this, use @Guide to tell the AI exactly what options to choose from:

```
@Generable
struct TransactionCategory {
    @Guide(description: "The category. Must be one of: Groceries,
Transport, Entertainment, Bills, Dining, Shopping, Health, or Other")
    let category: String

    @Guide(description: "How confident is this categorisation. Must be:
High, Medium, or Low")
    let confidence: String

    @Guide(description: "Type of merchant. Must be: Supermarket,
Restaurant, Online, Retail, Service, or Unknown")
    let merchantType: String

    @Guide(description: "A brief one-sentence reason for this
categorisation")
    let reason: String
}
```

Now the AI knows:

- For category: only return one of these 8 options
- For confidence: only return High, Medium, or Low
- And so on

Important: @Guide doesn't change what the AI decides — it changes how the AI expresses its decision.

Without @Guide : AI thinks "I'm very sure" → returns "Very Confident" or "95%" or "High"
With @Guide: AI thinks "I'm very sure" → returns "High" (because that's the only option for high confidence)

Is @Guide Required?

No! @Guide is optional. Your code will work without it. Use it when:

- Your UI depends on exact values (switch statements, filtering)
- You need consistent data for grouping or sorting
- You're building a production-grade app and need predictable, testable output values

Skip it when:

- You're just prototyping
- You only display the text to users
- Inconsistent formatting is acceptable

Step 5: Putting It Together — A Transaction Categoriser

Let's build a complete feature that categorises multiple transactions:

```
import SwiftUI
import FoundationModels

// What we want from the AI
@Generable
struct TransactionCategory {
    @Guide(description: "Must be: Groceries, Transport, Entertainment,
Bills, Dining, Shopping, Health, or Other")
    let category: String

    @Guide(description: "Must be: High, Medium, or Low")
    let confidence: String
}

// Our transaction model
struct Transaction: Identifiable {
    let id = UUID()
    let description: String
    let amount: Double
    var category: TransactionCategory?
    var isProcessing = false

    var formattedAmount: String {
        String(format: "£%.2f", amount)
    }
}

// The main view
struct TransactionListView: View {
    @State private var transactions = [
        Transaction(description: "Tesco Express", amount: 45.99),
        Transaction(description: "Uber", amount: 8.50),
        Transaction(description: "Netflix", amount: 15.99),
        Transaction(description: "Shell Petrol", amount: 65.00),
        Transaction(description: "Nando's", amount: 32.50)
    ]
    @State private var isProcessing = false

    var body: some View {
        NavigationStack {
            List(transactions) { transaction in
                HStack {
                    VStack(alignment: .leading) {
                        Text(transaction.description)
                            .font(.headline)
                        Text(transaction.formattedAmount)
                            .foregroundColor(.secondary)
```

```
                    }

                    Spacer()

                    if transaction.isProcessing {
                        ProgressView()
                    } else if let category = transaction.category {
                        Text(category.category)
                            .padding(6)
                            .background(Color.blue.opacity(0.1))
                            .cornerRadius(6)
                    }
                }
            }
            .navigationTitle("Transactions")
            .toolbar {
                Button("Categorise All") {
                    Task { await categoriseAll() }
                }
                .disabled(isProcessing)
            }
        }
    }

    func categoriseAll() async {
        isProcessing = true
        let session = LanguageModelSession()

        for index in transactions.indices {
            transactions[index].isProcessing = true

            do {
                let prompt = "Categorise this bank transaction:
\(transactions[index].formattedAmount) at
\(transactions[index].description)"

                let response = try await session.respond(
                    to: prompt,
                    generating: TransactionCategory.self
                )

                transactions[index].category = response.content
            } catch {
                print("Error: \(error)")
            }

            transactions[index].isProcessing = false
        }

        isProcessing = false
    }
}
```

When I tested this, the results were spot-on:

- Tesco → Groceries (High confidence)
- Uber → Transport (High confidence)
- Netflix → Entertainment (High confidence)
- Shell → Transport (High confidence)
- Nando's → Dining (High confidence)

The AI recognised all these well-known merchants and was confident in its categorisation.

Step 6: Adding Instructions

So far, we've been sending individual prompts. But what if you want the AI to behave a certain way for ALL requests in a session?

Use instructions:

```
let session = LanguageModelSession(
    instructions: "You are a UK banking assistant. Always use British
English. Be concise."
)

// Now every request in this session follows these rules
let response1 = try await session.respond(to: "Categorise: £45 at Tesco",
generating: TransactionCategory.self)
let response2 = try await session.respond(to: "Categorise: £12 at Uber",
generating: TransactionCategory.self)
```

Prompt vs Instructions

Concept	What It Is	When It Applies
prompt	The specific question you're asking	Just this one request
instructions	How the AI should behave	Every request in the session

Think of instructions as the AI's job description. The prompt is the task you give it each time.

Step 7: Tool Calling — When AI Needs Your Data

Here's the most powerful feature. So far, the AI has been working with information in the prompt. But what if a user asks:

- "How much did I spend on groceries this month?"

The AI doesn't have access to your transaction data. It only knows what you tell it in the prompt.

Tools: solve this, A tool lets the AI call back into your app to fetch data it needs.

```
struct SpendingTool: Tool {
    var name: String { "getSpending" }
    var description: String { "Get the user's spending total for a
category" }

    // Your app's data
    let transactions: [Transaction]

    // What the AI can ask for
    @Generable
    struct Arguments {
        @Guide(description: "Category to check: Groceries, Transport,
Entertainment, etc. Leave empty for all.")
        let category: String
    }
```

```
    // What happens when AI calls this tool
    func call(arguments: Arguments) async throws -> String {
        let categorised = transactions.filter { $0.category != nil }

        if !arguments.category.isEmpty {
            let filtered = categorised.filter {
                $0.category?.category.lowercased() ==
arguments.category.lowercased()
            }
            let total = filtered.reduce(0) { $0 + $1.amount }
            return "User spent £\(String(format: "%.2f", total)) on
\(arguments.category)"
        } else {
            let total = categorised.reduce(0) { $0 + $1.amount }
            return "User's total spending: £\(String(format: "%.2f",
total))"
        }
    }
}
```

Register the tool with your session:

```
let tool = SpendingTool(transactions: transactions)
let session = LanguageModelSession(
    instructions: "You are a helpful banking assistant.",
    tools: [tool]
)

let response = try await session.respond(to: "How much did I spend on
groceries?")
print(response.content)
// "You spent £45.99 on groceries."
```

What Happens Behind the Scenes

1. **User asks:** How much did I spend on groceries?
2. **AI realises:** I need spending data. I have a tool for that!
3. **AI calls your tool:** getSpending(category: "Groceries")`
4. **Your code runs:** Calculates total from transactions → returns "£45.99"
5. **AI responds:** "You spent £45.99 on groceries."

The AI decides **when** to call the tool and **what arguments** to pass. You just define what the tool does.

Step 8: Handling Invalid User Questions

There's an important issue to address. What happens if the user asks something unrelated?

- "What's the weather today?"
- "Tell me a joke"
- "asdfghjkl"

The AI might still try to call your spending tool and return random spending data — which is wrong!

The Problem

Without validation, the AI doesn't always know when NOT to use the tool. It might respond to "Hello" with "Your total spending is £417.52" — confusing and incorrect.

The Solution: Validate Before Calling AI
Add a simple keyword check before sending to the AI:

```
// Keywords that indicate a spending-related question
private let spendingKeywords = [
    "spend", "spent", "spending",
    "cost", "costs",
    "total", "totals",
    "how much",
    "expense", "expenses",
    "transaction", "transactions",
    "groceries", "transport", "entertainment", "bills", "dining",
"shopping", "health",
    "money", "budget",
    "category", "categories",
    "summary", "breakdown"
]
```

```
// Check if question is related to spending
private func isSpendingRelatedQuestion(_ question: String) -> Bool {
    let lowercased = question.lowercased()
    return spendingKeywords.contains { lowercased.contains($0) }
}

func askQuestion() async {
    // Validate BEFORE calling AI
    guard isSpendingRelatedQuestion(userQuestion) else {
        chatResponse = "I'm your banking assistant. I can help you with
questions about your spending and transactions. Try asking: 'How much did I
spend on groceries?' or 'What's my total spending?'"
        return
    }

    // Only call AI if question is valid
    let tool = SpendingTool(transactions: transactions)
    let session = LanguageModelSession(
        instructions: "You are a helpful banking assistant.",
        tools: [tool]
    )

    let response = try await session.respond(to: userQuestion)
    chatResponse = response.content
}
```

Now It Works Correctly

User Question	Response
How much on groceries?	You spent £113.29 on groceries
Total spending	Your total spending is £417.52
What's the weather?	'm your banking assistant. I can help you with questions about your spending...
Hello	'm your banking assistant. I can help you with

	questions about your spending..
asdfghjkl	'm your banking assistant. I can help you with questions about your spending...

Why Validate in Code, Not Just Instructions?

You might think: "Can't I just tell the AI in instructions to ignore unrelated questions?"

The problem is AI instructions aren't always reliable:

Too strict instructions:AI rejects even valid questions like "dining expenses"
Too loose instructions:AI answers "What's the weather?" with spending data

Validating in your code is more reliable — and is a critical best practice for any production-grade app. You control exactly what gets through to the AI.

Quick Reference:

Concept	What It Does	You Define	AI Decides
@Generable	Returns a struct instead of text	The properties you need	The values for each property
@Guide	Constrains what values AI can use	The allowed options	Which option fits best
prompt	The specific question	The question text	The answer
instructions	AI's behaviour for the session	The rules/personality	How to apply them
tools	Lets AI fetch data from your app	What data is available	When to fetch it

Wrapping Up

Apple's Foundation Models framework makes on-device AI practical for everyday iOS development. No cloud setup, no API keys, no privacy concerns.

The key concepts:

- Use @Generable to get structured data: you define the structure, AI fills the values
- Use @Guide to ensure consistent values: you define the options, AI picks the best fit
- Use instructions to set the AI's behaviour for a session
- Use tools when the AI needs data from your app
- Validate user input before calling AI: don't rely on AI alone to reject invalid questions

If you're building apps that handle sensitive data — banking, health, finance — this is exactly what you need. All the AI capabilities, none of the privacy headaches.

Give it a try. Start with a simple @Generable struct and see what you can build.

Tested with Xcode 26 and iOS 26. Requires a device with Apple Intelligence enabled.

Check out the code on https://github.com/AddikiPraveenKumar/foundation-models-ios-guide