Sample Code

Generate dynamic game content with guided generation and tools

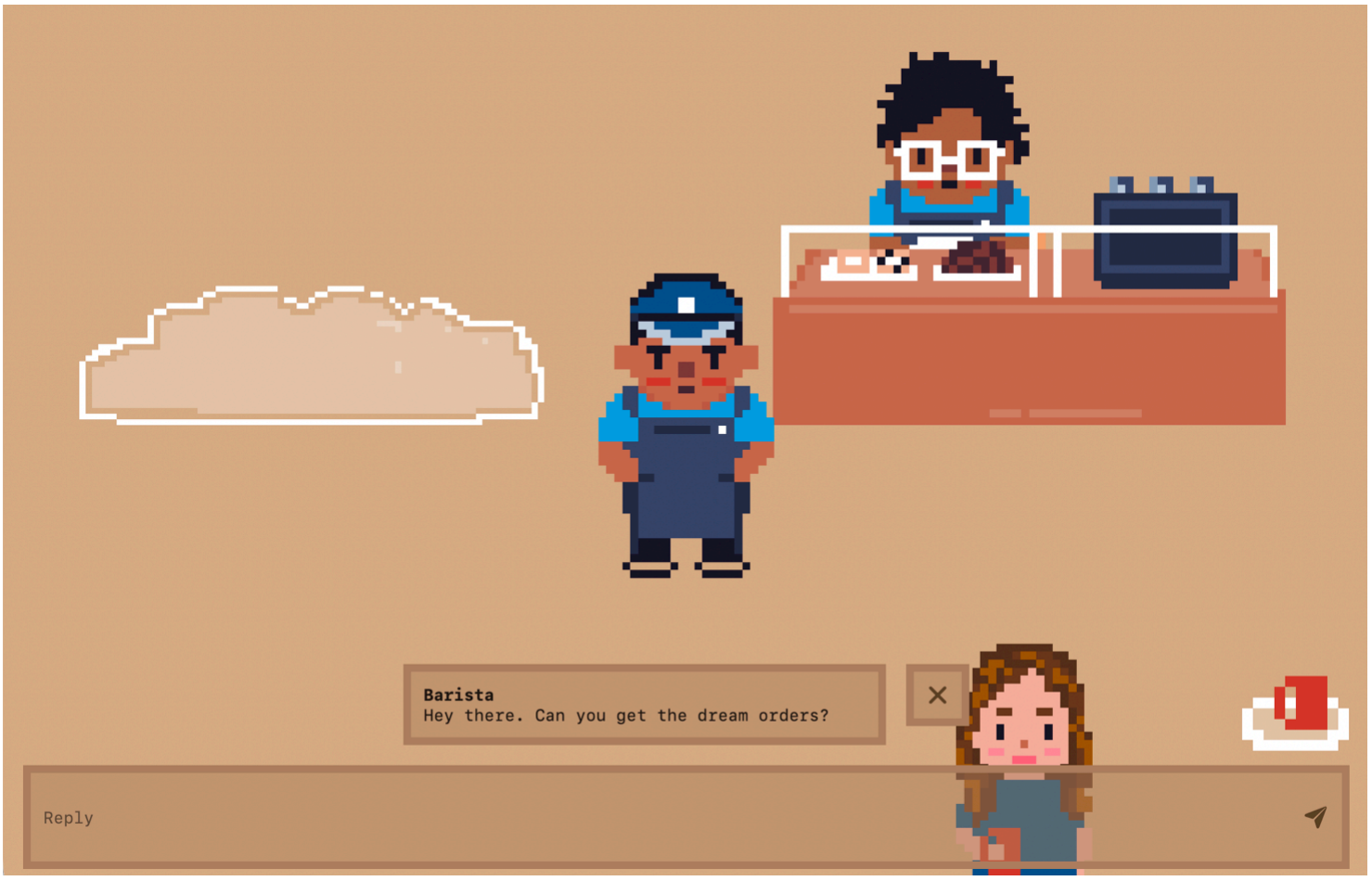
Make gameplay more lively with AI generated dialog and encounters personalized to the player.

Download

iOS 26.0+ | iPadOS 26.0+ | macOS 26.0+ | Xcode 26.0+

Overview

This sample code project demonstrates the Foundation Models framework and its ability to generate dynamic content for a game. Instead of using the same dialog script for customer encounters, the app dynamically generates dialog so that each time a player talks to a character, they can have a different conversation.



The game combines several framework capabilities — like guided generation and tool calling — to create dynamic, personalized gameplay experiences. You interact with both scripted characters, like the head barista, and procedurally generated customers, each with unique personalities, appearances, and coffee orders. As you serve customers, you can engage in conversations, take custom coffee orders, and receive feedback on your brewing skills — all powered by an on-device foundation model.

> **Note**
>
> This sample code project is associated with WWDC25 sessions 301: Deep Dive into the Foundation Models Framework.

Generate character dialog

The sample app generates dialog for characters by using `Character` to describe the character, like the barista:

```
struct Barista: Character {
    let id = UUID()
    let displayName = "Barista"
    let firstLine = "Hey there. Can you get the dream orders?"
```

```
    let persona = """
        Chike is the head barista at Dream Coffee, and loves serving up the perfect
        to all the dreamers and creatures in the dream realm. Today is a particularl
        Chike is happy to have the help of a new trainee barista named Player.
        """

    let errorResponse = "Maybe let's stop chatting? We've got coffee to serve."
}
```

A `persona` is a detailed description of the character that the model should pretend to be. The app uses a fixed error response when it encounters a generation error or content that the system blocks for safety.

The `DialogEngine` class manages conoversations for all characters in the game using `LanguageModelSession`. Each character maintains their own conversation session, allowing for persistent, contextual dialog that remembers previous interactions. When a conversation begins with a character, the dialog engine creates a new session with specific instructions that define the character's personality and role:

```
let instructions = """
    A multiturn conversation between a game character and the player of this game. \
    You are \(character.displayName). Refer to \(character.displayName) in the first
    (like "I" or "me"). You must respond in the voice of \(character.persona).\

    Keep your responses short and positive. Remember: Because this is the dream real
    everything is free at this coffee shop and the baristas are paid in creative inp

    You just said: "\(startWith)"
    """
```

When the player provides input text to *talk* to the character, the sample app uses the input as a prompt to the session. When generating a response, the dialog engine includes safety mechanisms to keep conversations on topic. It maintains block lists for words and phrases that characters shouldn't discuss, ensuring nonplayer characters (NPCs) focus on coffee-related topics. If the app generates content containing blocked terms, it automatically resets the conversation and provides the default error response for the character.

```
let response = try await session.respond(
    to: userInput
)
let dialog = response.content
```

```
// Verify whether the input contains any blocked words or phrases.
if textIsOK(dialog) {
    nextUtterance = dialog
    isGenerating = false
} else {
    nextUtterance = character.errorResponse
    isGenerating = false
    resetSession(character, startWith: character.resumeConversationLine)
}
```

If the output dialog fails the blocked phrases check, the model may break character or discuss something that's outside of the game world. To keep the dialog immersive, set `nextUtterance` to the character's fixed error response and reset the session.

Generate random encounters

The `EncounterEngine` creates unique customer encounters using the `Generable` protocol to generate structured content. Each encounter produces an NPC with a name, coffee order, and visual description.

```
@Generable
struct NPC: Equatable {
    let name: String
    let coffeeOrder: String
    let picture: GenerableImage
}
```

The process of generating an NPC uses a `LanguageModelSession` with a prompt that provide examples of the output format:

```
let session = LanguageModelSession {
    """
    A conversation between the Player and a helpful assistant. This is a fantasy
    RPG game that takes place at Dream Coffee, the beloved coffee shop of the
    dream realm. Your role is to use your imagination to generate fun game characte
    """
}
let prompt = """
    Create an NPC customer with a fun personality suitable for the dream realm. Have
    coffee. Here are some examples to inspire you:
```

```
    {name: "Thimblefoot", imageDescription: "A horse with a rainbow mane",
    coffeeOrder: "I would like a coffee that's refreshing and sweet like grass of a
    {name: "Spiderkid", imageDescription: "A furry spider with a cool baseball cap",
    coffeeOrder: "An iced coffee please, that's as spooky as me!"}
    {name: "Wise Fairy", imageDescription: "A blue glowing fairy that radiates wisdo
    coffeeOrder: "Something simple and plant-based please, that will restore my wise
    """

// Generate the NPC using the custom generable type.
let npc = try await session.respond(
    to: prompt,
    generating: NPC.self,
).content
```

Each generated NPC includes a `GenerableImage` that creates a visual representation of the character by using Image Playground. The image generation avoids human-like appearances, focusing instead on fantastical creatures, animals, and objects that fit the dream realm aesthetic. The `GenerableImage` class shows how to use `GenerationSchema` to describe the properties and guides of the object. This allows for creating dynamic schemas when all of the details of the generable type isn't known until runtime.

Use a language model to judge in-game creations

The game uses the on-device model to evaluate player performance through the `judgeDrink(drink:)` method in the encounter engine. When the player creates a coffee drink for a customer, the model assumes the customer's persona and provides feedback on whether the drink matches their original order.

The judging system creates a new `LanguageModelSession` that uses the specific customer's personality and preferences, and a prompt that provides the drink details for the model to evaluate:

```
let session = LanguageModelSession {
    """
    A conversation between a user and a helpful assistant. This is a fantasy RPG
    game that takes place at Dream Coffee, the beloved coffee shop of the dream
    realm. Your role is to pretend to be the following customer:
    \(customer.name): \(customer.picture.imageDescription)
    """
}
let prompt = """
    You have just ordered the following drink:
```

```
    \(customer.coffeeOrder)
    The barista has just made you this drink:
    \(drink)
    Does this drink match your expectations? Do you like it? You must respond
    with helpful feedback for the barista. If you like your drink, give it a
    compliment. If you dislike your drink, politely tell the barista why.
    """
return try await session.respond(to: prompt).content
```

The model then compares the player's creation against the customer's original order, providing contextual feedback that's authentic to the character's personality. This creates a dynamic evaluation system where the same drink might receive different reactions from different customers based on their unique preferences and personas.

Use tools to personalize game content

For customers that the sample generates, provide the dialog engine with custom tools, like `CalendarTool` to create more personalized interactions. This allows characters to reference the player's on-device information, making conversations feel more natural and connected to the player's actual life.

The `CalendarTool` integrates with EventKit to access the player's calendar events, and allows characters to reference real upcoming events that involve the customer's name if they are an attendee:

```
if let customer = character as? GeneratedCustomer {
    newSession = LanguageModelSession(
        tools: [CalendarTool(contactName: customer.displayName)],
        instructions: instructions
    )
}
```

The tool description tells the model what it uses the tool for:

```
description = """
    Get an event from the player's calendar with \(contactName). \
    Today is \(Date().formatted(date: .complete, time: .omitted))
    """
```

The sample app also provides a `ContactTool` that accesses the player's contacts to find names of people born in specific months. This allows the game to generate a coffee shop customer with names the player is familiar with.

```
let session = LanguageModelSession(
    tools: [contactsTool],
    instructions: """
        Use the \(contactsTool.name) tool to get a name for a customer.
        """
)
```

See Also

Tool calling

Expanding generation with tool calling

Build tools that enable the model to perform tasks that are specific to your use case.

`protocol Tool`

A tool that a model can call to gather information at runtime or perform side effects.