Article

Generating Swift data structures with guided generation

Create robust apps by describing output you want programmatically.

Overview

When you perform a request, the model returns a raw string in its natural language format. Raw strings require you to manually parse the details you want. Instead of working with raw strings, the framework provides guided generation, which gives strong guarantees that the response is in a format you expect.

To use guided generation, describe the output you want as a new Swift type. When you make a request to the model, include your custom type and the framework performs the work necessary to fill in and return an object with the parameters filled in for you. The framework uses constrained sampling when generating output, which defines the rules on what the model can generate. Constrained sampling prevents the model from producing malformed output and provides you with results as a type you define.

For more information about creating a session and prompting the model, see Generating content and performing tasks with Foundation Models.

Conform your data type to Generable

To conform your type to `Generable`, describe the type and the parameters to guide the response of the model. The framework supports generating content with basic Swift types like `Bool`, `Int`, `Float`, `Double`, `Decimal`, and `Array`. For example, if you only want the model to return a numeric result, call `respond(to:generating:includeSchemaInPrompt:options:)` using the type `Float`:

```
let prompt = "How many tablespoons are in a cup?"
let session = LanguageModelSession(model: .default)

// Generate a response with the type `Float`, instead of `String`.
let response = try await session.respond(to: prompt, generating: Float.self)
```

A schema provides the ability to control the values of a property, and you can specify guides to control values you associate with it. The framework provides two macros that help you with schema creation. Use `Generable(description:)` on structures, actors, and enumerations; and only use `Guide(description:)` with stored properties.

When you add descriptions to `Generable` properties, you help the model understand the semantics of the properties. Keep the descriptions as short as possible — long descriptions take up additional context size and can introduce latency. The following example creates a type that describes a cat and includes a name, an age that's constrained to a range of values, and a short profile:

```
@Generable(description: "Basic profile information about a cat")
struct CatProfile {
    // A guide isn't necessary for basic fields.
    var name: String

    @Guide(description: "The age of the cat", .range(0...20))
    var age: Int

    @Guide(description: "A one sentence profile about the cat's personality")
    var profile: String
}
```

> **Note**
>
> The model generates `Generable` properties in the order they're declared.

You can nest custom `Generable` types inside other `Generable` types, and mark enumerations with associated values as `Generable`. The `Generable` macro ensures that all associated and nested values are themselves generable. This allows for advanced use cases like creating complex data types or dynamically generating views at runtime.

Make a request with your custom data type

After creating your type, use it along with a `LanguageModelSession` to prompt the model. When you use a `Generable` type it prevents the model from producing malformed output and prevents the need for any manual string parsing.

```
// Generate a response using a custom type.
let response = try await session.respond(
    to: "Generate a cute rescue cat",
    generating: CatProfile.self
)
```

Define a dynamic schema at runtime

If you don't know what you want the model to produce at compile time use `DynamicGenerationSchema` to define what you need. For example, when you're working on a restaurant app and want to restrict the model to pick from menu options that a restaurant provides. Because each restaurant provides a different menu, the schema won't be known in its entirety until runtime.

```
// Create the dynamic schema at runtime.
let menuSchema = DynamicGenerationSchema(
    name: "Menu",
    properties: [
        DynamicGenerationSchema.Property(
            name: "dailySoup",
            schema: DynamicGenerationSchema(
                name: "dailySoup",
                anyOf: ["Tomato", "Chicken Noodle", "Clam Chowder"]
            )
        )

        // Add additional properties.
    ]
)
```

After creating a dynamic schema, use it to create a `GenerationSchema` that you provide with your request. When you try to create a generation schema, it can throw an error if there are conflicting property names, undefined references, or duplicate types.

```
// Create the schema.
let schema = try GenerationSchema(root: menuSchema, dependencies: [])
```

```
// Pass the schema to the model to guide the output.
let response = try await session.respond(
    to: "The prompt you want to make.",
    schema: schema
)
```

The response you get is an instance of `GeneratedContent`. You can decode the outputs from schemas you define at runtime by calling `value(_:forProperty:)` for the property you want.

See Also

Guided generation

`protocol` **`Generable`**

A type that the model uses when responding to prompts.