Sample Code

Adding and editing persistent data in your app

Create a data entry form for collecting and changing data managed by SwiftData.

Download

iOS 17.0+ | iPadOS 17.0+ | macOS 14.0+ | tvOS 17.0+ | Xcode 15.0+

Overview

Adding and editing data are fundamental features of data-driven apps, but how an app provides these features is unique to the app. This sample shows one approach, which is to use a data-entry form with SwiftData that lets someone add, edit, and store data about animals.

Define the data model

Before SwiftData can store data from your app, the app must define the data model that represents the data. SwiftData uses model classes to construct the schema of the data model. For example, the sample app stores data about animals, and groups those animals into categories. To define the schema for this data model, the sample defines two model classes: `Animal` and `AnimalCategory`.

The `Animal` model class stores information about an animal, like its name and diet. To persist instances of `Animal`, the class definition applies the `Model()` macro. This macro generates code at compile time that ensures the class conforms to the `PersistentModel` protocol and makes it possible for SwiftData to save animal data to a model container.

```
import SwiftData

@Model
```

```swift
final class Animal {
    var name: String
    var diet: Diet
    var category: AnimalCategory?

    init(name: String, diet: Diet) {
        self.name = name
        self.diet = diet
    }
}
```

The `AnimalCategory` model class stores information about an animal category, such as mammal or reptile. As with `Animal`, the `AnimalCategory` definition applies the `Model()` macro to ensure the class conforms to `PersistentModel` and to save the animal category data to a model container.

```swift
import SwiftData

@Model
final class AnimalCategory {
    @Attribute(.unique) var name: String
    // `.cascade` tells SwiftData to delete all animals contained in the
    // category when deleting it.
    @Relationship(deleteRule: .cascade, inverse: \Animal.category)
    var animals = [Animal]()

    init(name: String) {
        self.name = name
    }
}
```

The model class also has two properties:

`name`
: The name of the category. Each category name must be unique across all animal categories. To ensure this uniqueness, the model class applies the `Attribute(_:originalName:hashModifier:)` macro to the property with the option `unique`. This option ensures a property's value is unique across all models of the same type. For a complete list of options, see `Schema.Attribute.Option`.

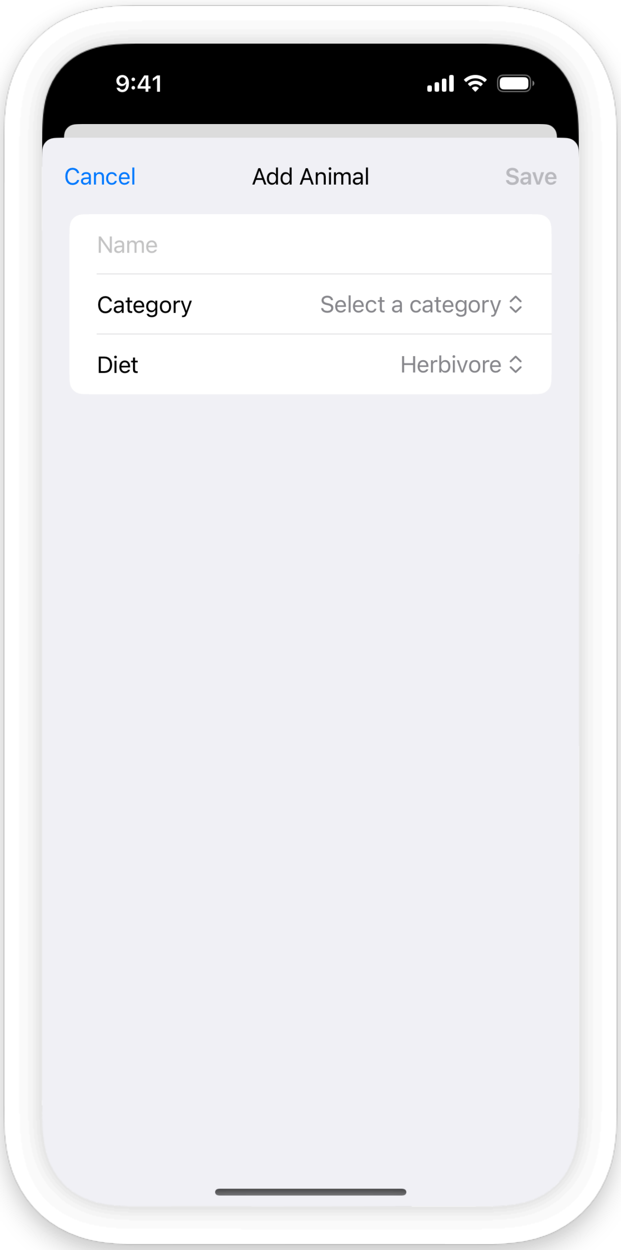
`animal`
: The list of animals contained in the category. The model class applies the `Relationship(_:deleteRule:minimumModelCount:maximumModelCount:originalName:inverse:`

hashModifier:) macro to this property to form a relationship between the model classes AnimalCategory and Animal. To learn more about the relationship, see Defining data relationships with enumerations and model classes.

Design the data editor

When deciding how people add and edit data in your app, consider the user experience. The sample app, for instance, lets someone add and edit information about animals using a custom data entry view, named AnimalEditor.

iOS iPadOS macOS tvOS



The design of AnimalEditor allows the app to use the same view for both adding new animals and editing existing ones. To provide this behavior, the editor declares the animal property as an optional Animal type. If animal is nil, a person using the editor is adding an animal; otherwise, the person is editing an existing animal. The editor makes the intention obvious by determining the title of the editor based on the value of `animal` in a computed property.

```
struct AnimalEditor: View {
    let animal: Animal?

    private var editorTitle: String {
        animal == nil ? "Add Animal" : "Edit Animal"
    }
    // ...
}
```

To enable editing the values of a new or existing animal, the editor defines state variables for each editable value. These state variables store the data that a person enters into the editor, separating what they enter from the data stored in `animal`. This separation ensures that SwiftData doesn’t save changes that a person makes until they’re ready to save those changes. This also gives them an opportunity to discard any changes they may have made to the data in the editor.

```
@State private var name = ""
@State private var selectedDiet = Animal.Diet.herbivorous
@State private var selectedCategory: AnimalCategory?

var body: some View {
    NavigationStack {
        Form {
            TextField("Name", text: $name)

            Picker("Category", selection: $selectedCategory) {
                Text("Select a category").tag(nil as AnimalCategory?)
                ForEach(categories) { category in
                    Text(category.name).tag(category as AnimalCategory?)
                }
            }

            Picker("Diet", selection: $selectedDiet) {
                ForEach(Animal.Diet.allCases, id: \.self) { diet in
                    Text(diet.rawValue).tag(diet)
                }
            }
        }
    }
}
```

The sample app takes this approach because it uses the autosave feature from SwiftData. The autosave feature automatically saves data changes made to model class instances, such as `animal`, instead of relying on the app to make explicit calls to the model context `save()` method. For more information about autosave, see `autosaveEnabled`.

Finally, to make the purpose of the editor clear to the person using it, `AnimalEditor` uses the `editorTitle` computed property to displays the title in the `principal` item section of the toolbar:

```
.toolbar {
    ToolbarItem(placement: .principal) {
        Text(editorTitle)
    }
}
```

Set default values

The `AnimalEditor` view declares its state variables with default values for a new animal, setting `name` to an empty string, `selectedDiet` to `herbivorous`, and leaving `selectedCategory` as `nil`. But the editor also supports editing an existing animal.

If someone edits an animal, the editor needs to show the values of the animal to edit, not the default values for the new animal. The view stores the animal to edit in the `animal` property. To show the current values of that animal, the editor applies the `onAppear(perform:)` modifier and copies the editable values from `animal` to the state variables:

```
.onAppear {
    if let animal {
        // Edit the incoming animal.
        name = animal.name
        selectedDiet = animal.diet
        selectedCategory = animal.category
    }
}
```

Save the data changes

To allow a person to save the changes they made in the editor, the editor provides a Save button in the toolbar:

```
ToolbarItem(placement: .confirmationAction) {
    Button("Save") {
        withAnimation {
            save()
            dismiss()
        }
    }
}
```

When a person clicks the Save button, it calls the editor's `save` method. If the person is editing an existing animal, `save` copies the values from the state variables to the instance of `Animal`. This directly updates the data that SwiftData manages, and because the app uses the autosave feature, SwiftData automatically saves the changes without calling the model context `save()` method.

```
private func save() {
    if let animal {
        // Edit the animal.
        animal.name = name
        animal.diet = selectedDiet
        animal.category = selectedCategory
    } else {
        // Add an animal.
        // ...
    }
}
```

When adding a new animal, the `save` function creates a new `Animal` instance, initializing it with the name and diet from the state variables. Then it sets the category and inserts the animal into the model context by calling the model context `insert(_:)` method:

```
private func save() {
    if let animal {
        // Edit the animal.
        // ...
    } else {
        // Add an animal.
        let newAnimal = Animal(name: name, diet: selectedDiet)
        newAnimal.category = selectedCategory
        modelContext.insert(newAnimal)
    }
}
```

After saving the data, the Save button's action closes the editor by calling `dismiss`.

> **Note**
>
> The `AnimalEditor` view retrieves the model context from the view's environment by creating a reference to the context with `@Environment(\.modelContext) private var modelContext`. For more information, see `modelContext`.

Discard the data changes

To discard changes that someone made, the editor provides a Cancel button in the toolbar:

```
ToolbarItem(placement: .cancellationAction) {
    Button("Cancel", role: .cancel) {
        dismiss()
    }
}
```

When a person clicks the Cancel button, the editor discards any changes made to the data by calling `dismiss` in the button’s action. This closes the editor without saving the changes.

See Also

Essentials

Preserving your app’s model data across launches

Describe your model classes to SwiftData using the framework’s macros, and store instances of those models so they exist beyond the app’s runtime.

{} Adopting SwiftData for a Core Data app

Persist data in your app intuitively with the Swift native persistence framework.

SwiftData updates

Learn about important changes to SwiftData.

Adopting inheritance in SwiftData

Add flexibility to your models using class inheritance.