Sample Code

Deleting persistent data from your app

Explore different ways to use SwiftData to delete persistent data.

Download

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

Overview

Data-driven apps typically provide a way for a person to delete data, and this sample is no exception. It shows three ways to remove data stored in a SwiftData model container:

- Swiping to delete
- Deleting with confirmation
- Deleting all

> **Note**
>
> If you want to learn how to add and edit data using SwiftData, see Adding and editing persistent data in your app.

Swipe to delete

The sample app shows a list of animals. A person using the app can delete an animal using a swipe gesture. For example, the following code adds the swipe-to-delete option to the `AnimalList` view by applying the `onDelete(perform:)` modifier to `ForEach`:

```
private struct AnimalList: View {
    @Environment(NavigationContext.self) private var navigationContext
    @Environment(\.modelContext) private var modelContext
    @Query(sort: \Animal.name) private var animals: [Animal]

    var body: some View {
        @Bindable var navigationContext = navigationContext
        List(selection: $navigationContext.selectedAnimal) {
            ForEach(animals) { animal in
                NavigationLink(animal.name, value: animal)
            }
            .onDelete(perform: removeAnimals)
        }
    }
}
```

The `onDelete(perform:)` modifier in the previous code calls the custom method `removeAnimals` to remove one or more animals from the list. This method receives an `IndexSet` which identifies the animals to delete. The method then iterates through the index set, deleting each animal using the `ModelContext` method `delete(_:)`.

```
private func removeAnimals(at indexSet: IndexSet) {
    for index in indexSet {
        modelContext.delete(animals[index])
    }
}
```

However, it's possible for a person using this sample to delete the selected animal, so the `removeAnimals` method needs to unselect the animal before deleting it. The updated version of `removeAnimals` uses the `persistentModelID` to determine whether the animal to delete is also the selected animal. If it is, the method sets the selected animal to `nil`.

```
private func removeAnimals(at indexSet: IndexSet) {
    for index in indexSet {
        let animalToDelete = animals[index]
        if navigationContext.selectedAnimal?.persistentModelID == animalToDelete.per
            navigationContext.selectedAnimal = nil
        }
        modelContext.delete(animalToDelete)
    }
}
```

The sample uses the SwiftData autosave feature, which is enabled by default when creating a `ModelContainer` using the `modelContainer(for:inMemory:isAutosaveEnabled:isUndoEnabled:onSetup:)` modifier. If this feature is disabled, the `removeAnimals` method needs to explicitly save the change by calling the `ModelContext` method `save()`; for example:

```
private func removeAnimals(at indexSet: IndexSet) {
    do {
        for index in indexSet {
            let animalToDelete = animals[index]
            if navigationContext.selectedAnimal?.persistentModelID == animalToDelete
                navigationContext.selectedAnimal = nil
            }
            modelContext.delete(animalToDelete)
        }
        try modelContext.save()
    } catch {
        // Handle error.
    }
}
```

> **Note**
>
> To disable the autosave feature, set the `isAutoSaveEnabled` parameter to `false` when creating the model container using the `modelContainer(for:inMemory:isAutosaveEnabled:isUndoEnabled:onSetup:)` modifier. You can also disable autosave by setting set the model context property `autosaveEnabled` to `false`.

Delete with confirmation

The sample app also lets a person delete the selected animal by clicking the Trash button that `AnimalDetailView` displays in its toolbar.

```
struct AnimalDetailView: View {
    var animal: Animal?
    @State private var isDeleting = false
    @Environment(\.modelContext) private var modelContext
    @Environment(NavigationContext.self) private var navigationContext

    var body: some View {
        if let animal {
```

```
            AnimalDetailContentView(animal: animal)
                .navigationTitle("\(animal.name)")
                .toolbar {
                    Button { isDeleting = true } label: {
                        Label("Delete \(animal.name)", systemImage: "trash")
                            .help("Delete the animal")
                    }
                }
        } else {
            ContentUnavailableView("Select an animal", systemImage: "pawprint")
        }
    }
}
```

The action for the button set the state variable `isDeleting` to `true`, which displays the delete confirmation alert described in the following code:

```
.alert("Delete \(animal.name)?", isPresented: $isDeleting) {
    Button("Yes, delete \(animal.name)", role: .destructive) {
        delete(animal)
    }
}
```

After confirming the delete request, the action for the confirmation button calls the custom `delete` method. This method sets the selected animal to `nil`, then deletes the animal from the model context by calling the `delete(_:)` method.

```
private func delete(_ animal: Animal) {
    navigationContext.selectedAnimal = nil
    modelContext.delete(animal)
}
```

If the SwiftData autosave feature was disabled, the `delete` method would need to explicitly save the change by calling the `ModelContext` method `save()`; for example:

```
private func delete(_ animal: Animal) {
    do {
        navigationContext.selectedAnimal = nil
        modelContext.delete(animal)
        try modelContext.save()
    } catch {
```

```
        // Handle error.
    }
}
```

Delete all

Deleting all items of a particular model type is less common in data driven apps, but there may be times when having this option is helpful. For example, the sample apps lets a person reload sample data that comes with the app. Reloading the sample data deletes all animal categories and animals from persistent storage.

To delete all items of a particular model type, use the `ModelContext` method `delete(model:where:includeSubclasses:)`. For example, the following code deletes all animal categories before reloading the sample data:

```
static func reloadSampleData(modelContext: ModelContext) {
    do {
        try modelContext.delete(model: AnimalCategory.self)
        insertSampleData(modelContext: modelContext)
    } catch {
        fatalError(error.localizedDescription)
    }
}
```

When deleting all animal categories, SwiftData also deletes all animals within those categories. SwiftData knows to perform this cascading delete because the relationship between `AnimalCategory` and `Animal` uses the `Schema.Relationship.DeleteRule.cascade` delete rule. (For a complete list of delete rules, see `Schema.Relationship.DeleteRule`.)

```
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
```

```
    }
}
```

To delete all persistent data from your app and not just data of a certain model type, use the `ModelContainer` method `deleteAllData()`.

See Also

Model life cycle

`class ModelContainer`

An object that manages an app's schema and model storage configuration.

`class ModelContext`

An object that enables you to fetch, insert, and delete models, and save any changes to disk.

Fetching and filtering time-based model changes

Track all inserts, updates, and deletes that occur in a data store and process them as a series of chronological transactions.

`struct HistoryDescriptor`

A type that describes the criteria, and, optionally, sort order, to use when fetching history data

Reverting data changes using the undo manager

Automatically record data change operations that people perform in your SwiftUI app, and let them undo and redo those changes.

Syncing model data across a person's devices

Add the required capabilities and define a compatible schema to enable SwiftData to automatically sync your app's model data using iCloud.

Concurrency support

Types you use to access model attributes and perform storage-related tasks in a safe and isolated way.