Article

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.

Overview

Most apps define a number of custom types that model the data it creates or consumes. For example, a travel app might define classes that represent trips, flights, and booked accommodations. Using SwiftData, you can quickly and efficiently persist that data so it's available across app launches, and leverage the framework's integration with SwiftUI to refetch that data and display it onscreen.

By design, SwiftData supplements your existing model classes. The framework provides tools such as macros and property wrappers that enable you to expressively describe your app's schema in Swift code, removing any reliance on external dependencies such as model and migration mapping files.

Turn classes into models to make them persistable

To let SwiftData save instances of a model class, import the framework and annotate that class with the `Model()` macro. The macro updates the class with conformance to the `PersistentModel` protocol, which SwiftData uses to examine the class and generate an internal schema. Additionally, the macro enables change tracking for the class by adding conformance to the `Observable` protocol.

```
import SwiftData

// Annotate new or existing model classes with the @Model macro.
```

```
@Model
class Trip {
    var name: String
    var destination: String
    var startDate: Date
    var endDate: Date
    var accommodation: Accommodation?
}
```

By default, SwiftData includes all noncomputed properties of a class as long as they use compatible types. The framework supports primitive types such as `Bool`, `Int`, and `String`, as well as complex value types such as structures, enumerations, and other value types that conform to the `Codable` protocol.

The code you write to define your model classes now serves as the source of truth for your app's model layer, and the framework uses that to keep the persisted data in a consistent state.

Customize the persistence behavior of model attributes

An *attribute* is a property of a model class that SwiftData manages. In most cases, the framework's default behavior for attributes is sufficient. However, if you need to alter how SwiftData handles the persistence of a particular attribute, use one of the provided schema macros. For example, you may want to avoid conflicts in your model data by specifying that an attribute's value is unique across all instances of that model.

To customize an attribute's behavior, annotate the property with the `Attribute(_:originalName:hashModifier:)` macro and specify values for the options that drive the desired behavior:

```
@Attribute(.unique) var name: String
```

Aside from enforcing unique constraints, `@Attribute` supports, among others, preserving deleted values, Spotlight indexing, and encryption. You can also use the `@Attribute` macro to correctly handle renamed attributes if you want to preserve the original name in the underlying model data.

When a model contains an attribute whose type is also a model (or a collection of models), SwiftData implicitly manages the relationship between those models for you. By default, the framework sets relationship attributes to `nil` after you delete a related model instance. To specify a different deletion rule, annotate the property with the `Relationship(_:deleteRule:minimumModelCount:maximumModelCount:originalName:inverse:hashModifier:)` macro. For example, you may want to delete any related accommodations when deleting a trip. For more information about delete rules, see `Schema.Relationship.DeleteRule`.

```
@Relationship(.cascade) var accommodation: Accommodation?
```

SwiftData persists all noncomputed attributes of a model by default, but you may not always want this to happen. For example, one or more properties on a class may only ever contain temporary data that doesn't need saving, such as the current weather at an upcoming trip's destination. In such scenarios, annotate those properties with the `Transient()` macro and SwiftData won't write their values to disk.

```
@Transient var destinationWeather = Weather.current()
```

Configure the model storage

Before SwiftData can examine your models and generate the required schema, you need to tell it — at runtime — which models to persist, and optionally, the configuration to use for the underlying storage. For example, you may want the storage to exist only in memory when running tests, or to use a specific CloudKit container when syncing model data across devices.

To set up the default storage, use the `modelContainer(for:inMemory:isAutosaveEnabled:isUndoEnabled:onSetup:)` view modifier (or the scene equivalent) and specify the array of model types to persist. If you use the view modifier, add it at the very top of the view hierarchy so all nested views inherit the properly configured environment:

```
import SwiftUI
import SwiftData

@main
struct TripsApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
                .modelContainer(for: [
                    Trip.self,
                    Accommodation.self
                ])
        }
    }
}
```

If you're not using SwiftUI, create a model container manually using the appropriate initializer:

```
import SwiftData

let container = try ModelContainer([
    Trip.self,
    Accommodation.self
])
```

> **Tip**
>
> If a model type contains a relationship, you may omit the destination model type from the array. SwiftData automatically traverses a model's relationships and includes any destination model types for you.

Alternatively, use `ModelConfiguration` to create custom storage. The type provides a number of options to configure including whether:

- the storage exists only in memory.
- the storage is read-only.
- the app uses a specific App Group to store its model data.

```
let configuration = ModelConfiguration(isStoredInMemoryOnly: true, allowsSave: false

let container = try ModelContainer(
    for: Trip.self, Accommodation.self,
    configurations: configuration
)
```

> **Important**
>
> Automatic iCloud sync relies on the presence of the CloudKit entitlement, and SwiftData uses the first container it finds in that entitlement. If your app needs a particular container, use an instance of `ModelConfiguration` to specify that container.

Save models for later use

To manage instances of your model classes at runtime, use a *model context* — the object responsible for the in-memory model data and coordination with the model container to successfully persist that data. To get a context for your model container that's bound to the main actor, use the `modelContext` environment variable:

```
import SwiftUI
import SwiftData

struct ContentView: View {
    @Environment(\.modelContext) private var context
}
```

Outside of a view, or if you're not using SwiftUI, access the same actor-bound context directly using the model container:

```
let context = container.mainContext
```

In both instances, the returned context periodically checks whether it contains unsaved changes, and if so, implicitly saves those changes on your behalf. For contexts you create manually, set the `autosaveEnabled` property to `true` to get the same behavior.

To enable SwiftData to persist a model instance and begin tracking changes to it, insert the instance into the context:

```
var trip = Trip(name: name,
                destination: destination,
                startDate: startDate,
                endDate: endDate)

context.insert(trip)
```

Following the insert, you can save immediately by invoking the context's `save()` method, or rely on the context's implicit save behavior instead. Contexts automatically track changes to their known model instances and include those changes in subsequent saves. In addition to saving, you can use a context to fetch, enumerate, and delete model instances. For more information, see `ModelContext`.

Fetch models for display or additional processing

After you begin persisting model data, you'll likely want to retrieve that data, materialized as model instances, and display those instances in a view or take some other action on them. SwiftData provides the `Query` property wrapper and the `FetchDescriptor` type for performing fetches.

To fetch model instances, and optionally apply search criteria and a preferred sort order, use `@Query` in your SwiftUI view. The `@Model` macro adds `Observable` conformance to your model

classes, enabling SwiftUI to refresh the containing view whenever changes occur to any of the fetched instances.

```
import SwiftUI
import SwiftData

struct ContentView: View {
    @Query(sort: \.startDate, order: .reverse) var allTrips: [Trip]

    var body: some View {
        List {
            ForEach(allTrips) {
                TripView(for: $0)
            }
        }
    }
}
```

Outside of a view, or if you're not using SwiftUI, use one of the two fetch methods on `ModelContext`. Each method expects an instance of `FetchDescriptor` containing a predicate and a sort order. The fetch descriptor allows for additional configuration that influences batching, offsets, and prefetching, among others.

```
let context = container.mainContext

let upcomingTrips = FetchDescriptor<Trip>(
    predicate: #Predicate { $0.startDate > Date.now },
    sortBy: [
        .init(\.startDate)
    ]
)
upcomingTrips.fetchLimit = 50
upcomingTrips.includePendingChanges = true

let results = context.fetch(upcomingTrips)
```

For more information about predicates, see `Predicate`.

See Also

Essentials

{} Adding and editing persistent data in your app

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

{} 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.