Sample Code

Adopting SwiftData for a Core Data app

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

Download

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

Overview

This sample project is designed to help you understand how to adopt SwiftData in an existing Core Data app. The SampleTrips app fetches and displays all upcoming trips from the store, and allows people to create or remove trips, and to add, update, or remove information from the itinerary for each trip. There are three versions of this app:

- A Core Data version that demonstrates Core Data best practices.
- A SwiftData version that shows the complete app conversion from Core Data to SwiftData.
- A coexistence version, where the sample app uses Core Data, and adds a widget extension that uses SwiftData. This version covers a scenario where you might want to adopt SwiftData incrementally, or for certain portions of your app.

Configure the sample code project

Open the sample code project in Xcode. Before building it, do the following:

1. Set the developer team for all targets to your team so Xcode automatically manages the provisioning profile. For more information, see Assign a project to a team.
2. Replace the App Group container identifier — `group.com.example.apple-samplecode.SampleTrips` — with one specific to your team for the entire project. The identifier points to an App Group container that the app and widget use to share data. You can search for `group`

`.com.example.apple-samplecode.SampleTrips` using the Find navigator in Xcode, and then change all of the occurrences. For more information, see Configuring app groups.

Adopt SwiftData

The SwiftData sample sets up the schema with Swift types that conform to the `PersistentModel` protocol, which captures information about the app's types, including properties and relationships. Each model file corresponds to an individual entity, with identical entity names, properties, and relationships as its Core Data counterpart.

Each model file in this sample uses the `Model()` macro to add necessary conformances for the `PersistentModel` and `Observable` protocols:

```
@Model class Trip {
    #Index<Trip>([\.name], [\.startDate], [\.endDate], [\.name, \.startDate, \.endDa
    #Unique<Trip>([\.name, \.startDate, \.endDate])

    @Attribute(.preserveValueOnDeletion)
    var name: String
    var destination: String

    @Attribute(.preserveValueOnDeletion)
    var startDate: Date

    @Attribute(.preserveValueOnDeletion)
    var endDate: Date

    @Relationship(deleteRule: .cascade, inverse: \BucketListItem.trip)
    var bucketList: [BucketListItem] = [BucketListItem]()

    @Relationship(deleteRule: .cascade, inverse: \LivingAccommodation.trip)
    var livingAccommodation: LivingAccommodation?
    ...
```

Additionally, the app sets up the container using `ModelContainer` to ensure that all views access the same `ModelContainer`.

```
.modelContainer(modelContainer)
```

Setting up the `ModelContainer` also creates and sets a default `ModelContext` in the environment. The app can access the `ModelContext` from any scene or view using an

environment property.

```
@Environment(\.modelContext) private var modelContext
```

Create a persisted data object

This app creates a new instance of a trip and inserts it into the `ModelContext` for persistence:

```
if newTripSegment == .personal {
    newTrip = PersonalTrip(name: name, destination: destination, startDate: startDat
} else if newTripSegment == .business {
    newTrip = BusinessTrip(name: name, destination: destination, startDate: startDat
} else {
    newTrip = Trip(name: name, destination: destination, startDate: startDate, endDa
}
modelContext.insert(newTrip)
```

Persist data

The app uses the SwiftData implicit save feature to persist data. This implicit save occurs on UI life cycle events and on a timer after the context changes. For more information about enabling autosave, see the `autosaveEnabled` property.

The app calls `delete(_:)` on the `ModelContext` with the instance to delete.

```
modelContext.delete(trip)
```

Fetch persisted data

This sample app fetches the complete list of upcoming trips by wrapping an array of trips in a `Query` macro, which fetches `Trip` objects from the container.

```
@Query(sort: \Trip.startDate, order: .forward)
var trips: [Trip]
```

This sample also fetches data by calling `fetch(_:)` on the `ModelContext` and passing in a `FetchDescriptor` that specifies both the entity to retrieve data from as well as a corresponding `Predicate` that specifies the conditions for the object to fetch.

```
var descriptor = FetchDescriptor<BucketListItem>()
let tripName = trip.name
descriptor.predicate = #Predicate { item in
    item.title.contains(searchText) && tripName == item.trip?.name
}
let filteredList = try? modelContext.fetch(descriptor)
```

Inheritance

The SwiftData-Inheritance version of the app extends the `Trip` class into two distinct kinds of Trips, `PersonalTrip` and `BusinessTrip`, building on the basic `Trip` model to include more specialized properties for different kinds of Trips.

```
class PersonalTrip: Trip {...}
```

Both `PersonalTrip` and `BusinessTrip` inherit basic properties from their superclass, `Trip`, while defining their own specialized properties, as shown in the following code. For instance, `PersonalTrip`, has an additional property that describes the reason for the trip.

```
init(name: String, destination: String, startDate: Date = .now, endDate: Date = .dis
    self.reason = reason
    super.init(name: name, destination: destination, startDate: startDate, endDate:
}
```

Coexistence between Core Data and SwiftData

The coexistence version of the app has two persistence stacks: a Core Data persistence stack for the host app, and a SwiftData persistence stack for the widget extension. Both stacks write to the same store file.

Namespace models

The namespaces in the coexistence sample use the pre-existing `NSManagedObject`-based entity subclasses so that they don't conflict with the SwiftData classes. Note that this refers to the class name, not the entity name.

```
class CDTrip: NSManagedObject {...}
```

The sample then refers to the entity as `CDTrip` when accessing it in the Core Data host app. For instance, when adding a new `Trip`:

```
let newTrip = CDTrip(context: viewContext)
```

Share the same store file

This sample ensures that both the Core Data and SwiftData persistent stacks write to the same store file by setting the persistent store URL for the container description:

```
if let description = container.persistentStoreDescriptions.first {
    description.url = url
    ...
}
```

Additionally, the coexistence sample must set the `NSPersistentHistoryTrackingKey`. Although SwiftData enables persistent history tracking automatically, Core Data does not, so the app enables persistent history manually.

```
description.setOption(true as NSNumber, forKey: NSPersistentHistoryTrackingKey)
```

By default, SwiftData behaves in the following way when determining where it persists data:

- It persists data store to the app's Application Support directory.
- This sample app uses App Groups to access shared containers and share data between the SwiftData widget extension and the Core Data host app. For an app that has the `App Groups Entitlement`, it persists the data store to the root directory of the app group container. For apps that evolve from a version that doesn't have any app group container to a version that has one, SwiftData copies the existing store to the app group container.

In this sample, the main app and widget share the same store via an app group container, and the store is located in the default location in the app group container. To ensure SwiftData accesses the same store, the main app and widget both share the `ModelContainer`.

Detect relevant changes by consuming the SwiftData history

In the SwiftData version, people can confirm the living accommodation for the current trip by tapping the Accommodation button in the widget. The widget then updates `Living`

`Accommodation.isConfirmed` in SwiftData, and the main app, when entering the foreground, detects the changes and annotates the trip with a blue dot to indicate that it has unread changes.

There are multiple options for the main app to detect the changes from the widget:

1. Adding a key value pair to the shared `UserDefaults` (`init(suiteName:)` for the widget and the main app to share the changes.
2. Adding a new attribute in `Trip` so the widget can mark the trip as "unread" when changing the living accommodation status.
3. Consuming the history of the store, which SwiftData generates by default, and picking up the relevant changes from there.

The first option introduces a new storage, and hence needs to maintain the consistency between SwiftData and the shared `UserDefaults`. The second option is easier to implement, but introduces and maintains a new attribute, which is redundant and consumes extra storage space; for real-world apps that manage more complicated changes and have larger data set, that may not be the favorite approach.

This sample chooses to detect the changes with the third option. To do so, it sets up a `HistoryDescriptor<DefaultHistoryTransaction>` with a history token (`DefaultHistoryToken`) and calls `fetchHistory(_:)` to retrieve the history transactions (`DefaultHistoryTransaction`) after the token, as shown in the following code:

```
private func findTransactions(after historyToken: DefaultHistoryToken?, author: Stri
```

After getting the transactions, it uses the following code to find the trips that have living accommodation changes:

```
private func findTrips(in transactions: [DefaultHistoryTransaction]) -> (Set<Trip>,
```