SwiftData / ModelContext

Class

ModelContext

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

iOS 17.0+ | iPadOS 17.0+ | Mac Catalyst 17.0+ | macOS 14.0+ | tvOS 17.0+ | visionOS 1.0+ | watchOS 10.0+ | Swift 5.9+

```
class ModelContext
```

Mentioned in

Preserving your app’s model data across launches

Overview

A model context is central to SwiftData as it’s responsible for managing the entire lifecycle of your persistent models. You use a context to insert new models, track and persist changes to those models, and to delete those models when you no longer need them. A context understands your app’s schema but doesn’t know about any individual models until you tell it to fetch some from the persistent storage or populate it with new models. Afterwards, any changes made to those models exist only in memory until the context implicitly writes them to the persistent storage, or you manually invoke `save()`. For more information about implicit writes, see `autosaveEnabled`.

If your app’s schema describes relationships between models, you don’t need to manually insert each model into the context when you first create them. Instead, create the graph of related models and insert only the graph’s root model into the context. The context recognizes the hierarchy and automatically handles the insertion of the related models. The same behavior applies even if the graph contains both new and existing models.

A model context depends on a model container for knowledge about your app's schema and persistent storage. After you attach a container to your app's window group or view hierarchy, an associated context becomes available in the SwiftUI environment. This context is bound to the main actor and the framework configures the context to implicitly save future model changes. The `Query()` macros use the same context to perform their fetches.

```
struct LastModifiedView: View {
    @Environment(\.modelContext) private var modelContext

}
```

Important

If you don't explicitly attach a model container, the environment provides a context bound to an in-memory, schema-less container. Any attempt to insert a model into this context causes the framework to throw an error, and any fetches you run will return empty results.

After you establish access to a model context, use that context's `insert(_:)` and `delete(_:)` methods to add and remove models. You can also delete several models at once using `delete(model:where:includeSubclasses:)`. There isn't a corresponding method to update a model because the context automatically tracks all changes to its known models. Use the `hasChanges` property to determine if the context has unsaved changes, and call `rollback()` to discard any pending inserts and deletes and any restore changed models to their most recent saved state.

Although you fetch models primarily with the `Query()` macro (and its variants), you can use a model context to perform almost identical fetches. For example, use the `fetch(_:)` and `fetch(_:batchSize:)` methods to retrieve all models of a certain type that match a set of criteria. And use `fetchCount(_:)` to determine the number of models that match some criteria without the overhead of fetching the models themselves. If you need to be able to identify models that match some criteria but don't require all of the associated data, use `fetchIdentifiers(_:)` and `fetchIdentifiers(_:batchSize:)` to retrieve only those models' persistent identifiers.

A model context posts a `willSave` notification before it attempts a save operation, and a `didSave` notification immediately after that operation succeeds. Subscribe to one, or both, of these notifications if your app needs to be aware of these events. The `didSave` notification provides additional information about any inserted, updated, and deleted models.

```
struct LastModifiedView: View {
    @Environment(\.modelContext) private var context
    @State private var lastModified = Date.now
    private var didSavePublisher: NotificationCenter.Publisher {
        NotificationCenter.default
            .publisher(for: ModelContext.willSave, object: context)
    }

    var body: some View {
        Text(lastModified.formatted(date: .abbreviated, time: .shortened))
            .onReceive(didSavePublisher) { _ in
                lastModified = Date.now
            }
    }
}
```

> **Note**
>
> To avoid receiving unwanted or unexpected notifications, always specify the model context as the `object` parameter when creating a publisher.

Topics

Creating a model context

`init(ModelContainer)`

Creates a context that belongs to the specified model container.

`class ModelContainer`

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

Fetching models

`func fetch<T>(FetchDescriptor<T>) throws -> [T]`

Returns an array of typed models that match the criteria of the specified fetch descriptor.

`func fetch<T>(FetchDescriptor<T>, batchSize: Int) throws -> FetchResultsCollection<T>`

Returns a collection of typed models, in batches, which match the criteria of the specified fetch descriptor.

`func fetchCount<T>(FetchDescriptor<T>) throws -> Int`

Returns the number of models that match the criteria of the specified fetch descriptor.

`struct FetchDescriptor`

A type that describes the criteria, sort order, and any additional configuration to use when performing a fetch.

`struct FetchResultsCollection`

A collection that efficiently provides the results of a completed fetch.

`func enumerate<T>(FetchDescriptor<T>, batchSize: Int, allowEscapingMutations: Bool, block: (T) throws -> Void) throws`

Runs a closure for each model that matches the criteria of the specified fetch descriptor.

`func model(for: PersistentIdentifier) -> any PersistentModel`

Returns the persistent model for the specified identifier.

`func registeredModel<T>(for: PersistentIdentifier) -> T?`

Returns the typed model for the specified identifier.

Inserting models

`var insertedModelsArray: [any PersistentModel]`

The array of inserted models that the context is yet to persist.

`func insert<T>(T)`

Registers the specified model with the context so it can include the model in the next save operation.

Modifying models

`var hasChanges: Bool`

A Boolean value that indicates whether the context has unsaved changes.

`var changedModelsArray: [any PersistentModel]`

The array of registered models that have unsaved changes.

Deleting models

`var deletedModelsArray: [any PersistentModel]`

The array of registered models that the context will remove from the persistent storage during the next save operation.

`func delete<T>(T)`

Removes the specified model from the persistent storage during the next save operation.

`func delete<T>(model: T.Type, where: Predicate<T>?, includeSubclasses: Bool) throws`

Removes each model satisfying the given predicate from the persistent storage during the next save operation.

Persisting unsaved changes

`var autosaveEnabled: Bool`

A Boolean value that indicates whether the context should automatically save any pending changes when certain events occur.

`func save() throws`

Writes any pending inserts, changes, and deletes to the persistent storage.

`func transaction(block: () throws -> Void) throws`

Runs the provided closure, and once it finishes, writes any pending inserts, changes, and deletes to the persistent storage.

`func rollback()`

Discards pending inserts and deletes, restores changed models to their most recent committed state, and empties the undo stack.

Fetching only persistent identifiers

`func fetchIdentifiers<T>(FetchDescriptor<T>) throws -> [PersistentIdentifier]`

Returns an array of persistent identifiers, where each identifier represents a single model that satisfies the criteria of the specified fetch descriptor.

`func fetchIdentifiers<T>(FetchDescriptor<T>, batchSize: Int) throws -> FetchResultsCollection<PersistentIdentifier>`

Returns a collection of persistent identifiers, in batches, where each identifier represents a single model that satisfies the criteria of the specified fetch descriptor.

Accessing the container

`var container: ModelContainer`

The context’s model container.

Performing undo and redo

`func processPendingChanges()`

Tells the undo manager to record any changes made to the context’s registered models.

`var undoManager: UndoManager?`

The object that provides undo support for the context.

Registering for notifications

`static let willSave: Notification.Name`

A notification that posts when the context is about to process pending inserts, changes, and deletes.

`static let didSave: Notification.Name`

A notification that posts when the context finishes processing pending inserts, changes, and deletes.

`enum NotificationKey`

Describes the data in the user info dictionary of a notification sent by a model context.

Debugging contexts

`var debugDescription: String`

A textual representation of the context, suitable for debugging.

Instance Properties

`var author: String?`

`var editingState: EditingState`

Instance Methods

`func deleteHistory<T>(HistoryDescriptor<T>) throws`

`func fetchHistory<T>(HistoryDescriptor<T>) throws -> [T]`

Relationships

Conforms To

`Equatable`, `SendableMetatype`

See Also

Model life cycle

`class ModelContainer`

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

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

{} Deleting persistent data from your app

Explore different ways to use SwiftData to delete persistent 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.