


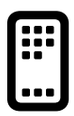


Core Data with SwiftUI

Core Data for Data Persistence

What is Core Data?

- Apple's framework for managing model layer objects
- Works across iOS, macOS, watchOS, and tvOS
- Not just a database, but a complete data modelling solution
- Provides features like persistence, undo/redo, and more

Core Data Stack

When you want to access data, you ask the Persistent Store Coordinator to find Persistent Store to retrieve the Managed Object Context.

Is Core Data Stack an ORM?

Core Data is not strictly an ORM (Object-Relational Mapping) tool, but it shares some similarities:

- It provides an object-oriented interface to your data.
- It can use SQLite as its backing store (but isn't limited to it).
- It handles the mapping between objects in your code and data in the store.

However, Core Data is more accurately described as an object graph and persistence framework. It offers features beyond typical ORMs, such as change tracking, undo/redo, and more.

Core Data: Persistent Store Types

SQLite Store

Type: NSSQLiteStoreType

Characteristics: Default option, efficient for large data sets, supports migrations.

Use Case: Most iOS apps, especially those with substantial data or complex relationships.

Binary Store

Type: NSBinaryStoreType
Characteristics: Entire store is loaded into memory, faster for smaller datasets.
Use Case: Apps with smaller data sets that need quick access and don't require incremental saving.

In-Memory Store

Type: NSInMemoryStoreType
Characteristics: Data exists only in RAM, lost when app terminates.
Use Case: Temporary data storage, caching, unit testing

4. XML Store (macOS only)

Type: NSXMLStoreType
Characteristics: Data stored in XML format, human-readable.
Use Case: macOS apps where data interoperability or human-readability is important.

```
let container = NSPersistentContainer(name: "MyDataModel")
let storeDescription = container.persistentStoreDescriptions.first
storeDescription?.type = NSSQLiteStoreType // or other store types
container.loadPersistentStores { (storeDescription, error) in
    // Handle errors
}
```

Setting Up Core Data in SwiftUI

```
@main
struct PersonalJournalApp: App {
    let persistenceController = PersistenceController.shared

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environment(\.managedObjectContext,
                    persistenceController.container.viewContext)
        }
    }
}
```

Fetching Data with SwiftUI

```
struct ContentView: View {
    @Environment(\.managedObjectContext) private var viewContext

    @FetchRequest(
        sortDescriptors: [NSSortDescriptor(keyPath: \Item.timestamp,
            ascending: true)],
        animation: .default)
    private var items: FetchedResults<Item>
```

SwiftUI makes fetching data from Core Data easy with the @FetchRequest property wrapper. This setup automatically updates our view when the underlying data changes.

Creating and Saving Data

```
let newItem = Item(context: viewContext)
newItem.timestamp = Date()

do {
    try viewContext.save()
} catch {
    let nsError = error as NSError
    fatalError("Unresolved error \(nsError), \(nsError.userInfo)")
}
```

Creating and saving data with Core Data involves creating a new managed object, setting its properties, and then saving the context.

Updating and Deleting Data

```
// Updating
item.timestamp = Date()

// Deleting
viewContext.delete(item)

// Don't forget to save the context after updates or deletions
try? viewContext.save()
```

Updating data is as simple as modifying the properties of a managed object. Deleting involves calling the delete method on the context with the object you want to remove.