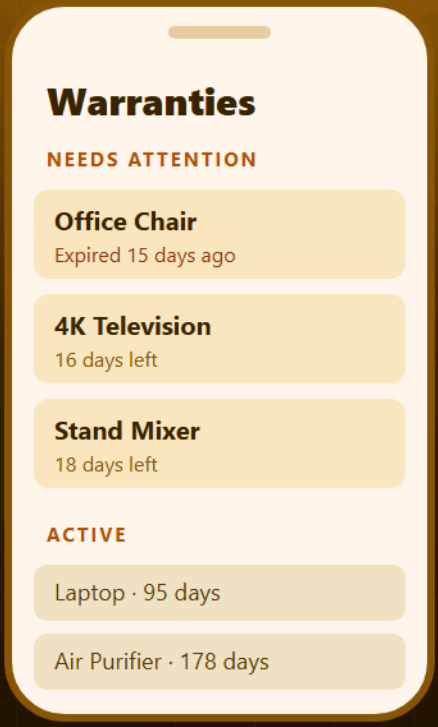



```swift
@Model final class Item {
  var name: String = ""
  var purchasePriceCents: Int?
  @Relationship(deleteRule: .cascade)
  var warranty: Warranty?
}
```

```swift
@ModelActor actor ItemImporter
static var stages: [MigrationStage]
cloudKitDatabase: .private(...)
SecItemAdd(query, nil)
```



A COMPLETE, PRACTICAL GUIDE

iOS Data Persistence

SwiftData, Core Data, CloudKit Sync, Files, and the Keychain

Models · Queries · Migrations · ModelActor · History
Core Data · iCloud Sync · CKSyncEngine · App Groups · Testing

CHOOSE · MODEL · MIGRATE · SYNC · TEST

THENEXBOOKS

iOS Data Persistence

SwiftData, Core Data, CloudKit Sync, Files, and the Keychain

Choose the right storage, model it well, migrate it safely, sync it, and test it — built step by step

First Edition

THENEXBOOKS

iOS Data Persistence
SwiftData, Core Data, CloudKit Sync, Files, and the Keychain

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Apple's frameworks, tools, and services change between releases; APIs, Xcode menus, availability, and iCloud behavior can differ from what is described here. Always consult Apple's current documentation for the versions you target, and back up real user data before running migrations or deleting stores.

Apple, iPhone, iPad, iOS, iPadOS, macOS, Xcode, Swift, SwiftUI, SwiftData, Core Data, CloudKit, iCloud, Keychain, Instruments, TestFlight, and App Store are trademarks of Apple Inc., registered in the U.S. and other countries and regions. SQLite is in the public domain; GRDB.swift and other third-party libraries are the property of their respective authors. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Apple Inc. All user-interface sketches in this book are original illustrations.

Shelfmate, the home inventory app used as the running example in this book, is fictional, as are its rooms, items, prices, serial numbers, warranty providers, the Coverwell Insurance insurer, and all sample data. Identifiers and web addresses use reserved example domains.

Preface

Every app that matters remembers something. A setting the user chose yesterday, a document they spent an hour on, a login token, a list of thousands of records, a photo, a sync state shared with their other devices. When that data is stored well, nobody notices. When it is stored badly, users lose their work, the app slows to a crawl as the list grows, an update crashes on launch because the database could not migrate, or two devices quietly disagree about what is true.

Apple gives iOS developers a rich set of storage tools: UserDefaults and `@AppStorage`, the file system with Codable, the Keychain, SwiftData, Core Data, CloudKit and its higher-level sync containers, App Groups, and SQLite underneath it all. Each tool is excellent at some jobs and wrong for others, and the documentation rarely tells you which is which, how they fit together, or what happens on the day your schema has to change.

This book is about getting local data right. We choose the correct tool for each kind of data, model it carefully, query it efficiently, move work off the main actor safely under Swift 6, evolve schemas without losing anything, sync with iCloud, share with widgets, protect and back up what users trust us with, and prove it all with tests. We go deep on both SwiftData and Core Data, because real teams maintain both — and we finish by migrating from one to the other.

Who This Book Is For

- **iOS developers who know Swift and SwiftUI basics** and now need persistence that holds up in production, not just in a demo.
- **Developers adopting SwiftData** who want to understand relationships, predicates, migrations, `ModelActor`, history, and CloudKit constraints before they ship.
- **Teams maintaining Core Data apps** who need a solid grasp of contexts, batch operations, persistent history, staged migrations, and a safe path toward SwiftData.
- **Anyone adding iCloud sync, widgets, or offline data** to an existing app and wanting to avoid the classic mistakes.

We assume you can build a simple SwiftUI app and read `async` / `await` code. We do not re-teach Swift syntax or SwiftUI layout; every page is about data.

What You Will Learn

- How to choose between UserDefaults, files, the Keychain, SwiftData, Core Data, SQLite, and CloudKit — and how to combine them in one app.
- How the app sandbox works: Documents, Application Support, Caches, temporary files, backups, and App Group containers.
- How to store preferences with `@AppStorage`, documents and backups with Codable and atomic writes, and secrets with Keychain Services.

- How to model data in SwiftData: attributes, relationships, delete rules, `@Query`, `FetchDescriptor`, `#Predicate`, editing, undo, and large binary data.
- How to run SwiftData safely in the background with `@ModelActor` under Swift 6, version schemas with `VersionedSchema` and migration plans, track changes with the history API, and keep it fast.
- How Core Data really works: the model editor, `NSPersistentContainer`, contexts and concurrency, fetched results controllers, batch requests, persistent history, and lightweight, staged, and custom migrations.
- How to sync with iCloud: CloudKit fundamentals, `NSPersistentCloudKitContainer`, SwiftData with CloudKit and its model rules, and custom sync with `CKSyncEngine`.
- How to share data with widgets and App Intents through App Groups, use SQLite directly when you need to, protect and back up data, recover from failures, test persistence thoroughly, and migrate from Core Data to SwiftData.

How This Book Is Organized

Part I — Foundations of Local Data (Chapters 1–5) surveys the storage landscape and covers the sandbox, UserDefaults and `@AppStorage`, files and Codable, and the Keychain. **Part II — SwiftData Essentials** (Chapters 6–10) builds Shelfmate's data model, queries, editing screens, and photo storage. **Part III — SwiftData in Production** (Chapters 11–14) covers background work with `ModelActor`, schema migrations, change history, and performance. **Part IV — Core Data in Depth** (Chapters 15–19) rebuilds the same data layer on Core Data and covers contexts, fetching, batch operations, persistent history, and migrations. **Part V — Sync and Sharing** (Chapters 20–24) covers CloudKit, syncing Core Data and SwiftData through iCloud, custom sync with `CKSyncEngine`, and sharing data with widgets and extensions. **Part VI — Production Data** (Chapters 25–28) covers SQLite directly, data protection, backups and recovery, testing, and migrating from Core Data to SwiftData. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Shelfmate

Throughout the book we build the data layer of **Shelfmate**, a fictional home inventory app. Shelfmate keeps track of the rooms in a home, the items in each room, what they cost, their serial numbers, photos, tags, and warranties — exactly the information you need after a burst pipe or a break-in, or simply when a warranty is about to run out. The sample home has five rooms and fifteen items worth $7,771.46 in total, with ten warranties, three of which need attention on the book's reference date of March 15, 2026. The data is small enough to check by hand, so every number in the book can be verified.

The main Shelfmate project uses SwiftData. In Part IV we work in **ShelfmateCD**, a Core Data edition of the same app with the same data, so you can compare the two frameworks line by line; in Chapter 28 we migrate ShelfmateCD's store to SwiftData. Along the way Shelfmate gains settings, file-protected JSON backups, a Keychain-stored insurer token, background CSV import, a versioned schema, iCloud sync, a warranty widget, and a test suite.

Versions and Conventions

Code is written for the **Swift 6 language mode** with complete data-race checking, using recent versions of Xcode, with a deployment target of **iOS 18**. SwiftData requires iOS 17 or later, and several features we use — such as the history API, `#Unique`, and `#Index` — require iOS 18. Features introduced with **iOS 26** are marked as such, guarded with availability checks, and kept out of the main code path. Apple's frameworks evolve every year, so when a detail may have changed, the text says so; check Apple's current documentation for the release you target.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause data loss, crashes, or security problems.

Swift code appears in `monospace` with the file it belongs to shown above it, such as `Shelfmate/Model/Item.swift`. Long lines are wrapped to fit the page. Where code prints something, we show what you should see, which may differ slightly in formatting on your machine. Raw SQL examples in Chapter 25 were run against the book's sample database. Money is stored as whole cents and displayed in US dollars.

Contents

Preface

PART I — FOUNDATIONS OF LOCAL DATA

1 The iOS Storage Landscape: Choosing the Right Tool for Every Kind of Data

2 The Sandbox, Containers, and FileManager

3 UserDefaults and @AppStorage: Preferences Done Right

4 Files and Codable: JSON, Property Lists, Atomic Writes, and Backups

5 The Keychain: Storing Secrets Safely

PART II — SWIFTDATA ESSENTIALS

6 SwiftData Fundamentals: @Model, ModelContainer, and ModelContext

7 Modeling Shelfmate: Attributes, Relationships, and Delete Rules

8 Querying with @Query, FetchDescriptor, and #Predicate

9 Editing Data in SwiftUI: Forms, Validation, Undo, and Saving

10 Photos and Large Data: External Storage, Files, and Thumbnails

PART III — SWIFTDATA IN PRODUCTION

11 Background Work with ModelActor and Swift 6 Concurrency

12 Schema Versioning and Migrations with VersionedSchema

13 Tracking Changes: SwiftData History and Keeping Processes in Sync

14 SwiftData Performance: Efficient Fetches, Indexes, Uniqueness, and Batch Work

PART IV — CORE DATA IN DEPTH

15 Core Data Fundamentals: The Model Editor, Entities, and NSPersistentContainer

16 Managed Object Contexts and Concurrency

17 Fetch Requests, NSFetchedResultsController, and @FetchRequest

18 Batch Operations, Persistent History, and Core Data Performance

19 Core Data Migrations: Lightweight, Staged, and Custom

PART V — SYNC AND SHARING

20 CloudKit Essentials for Persistence: Containers, Databases, Records, and Accounts

21 Syncing Core Data with NSPersistentCloudKitContainer

22 SwiftData and CloudKit: Model Rules, Setup, and Sync Behavior

23 Custom Sync with CKSyncEngine

24 App Groups: Sharing Data with Widgets, App Intents, and Extensions

PART VI — PRODUCTION DATA

25 SQLite Directly: The C API, Verifiable SQL, and When to Reach for GRDB

26 Data Protection, Backups, Privacy, and Recovering from Failure

27 Testing Persistence: In-Memory Stores, Swift Testing, and Migration Tests

28 Migrating from Core Data to SwiftData, and the Complete Shelfmate

APPENDICES

A Persistence Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations of Local Data

Choosing the right storage for every kind of data, understanding the app sandbox and its directories, storing preferences with UserDefaults and @AppStorage, writing documents and backups with Codable and atomic file writes, and keeping secrets in the Keychain.

1 · The iOS Storage Landscape

2 · The Sandbox, Containers, and FileManager

3 · UserDefaults and @AppStorage

4 · Files and Codable

5 · The Keychain

CHAPTER 1 · PART I — FOUNDATIONS OF LOCAL DATA

The iOS Storage Landscape: Choosing the Right Tool for Every Kind of Data

An iOS app rarely has one kind of data. It has preferences, documents, secrets, caches, structured records, and data that must follow the user to another device. This chapter maps each kind of data to the storage technology that fits it, introduces Shelfmate, the home inventory app we build throughout the book, and sets out the conventions every later chapter relies on.

What "Persistence" Really Means on iOS

In everyday conversation, "persistence" means "the data is still there next time." On iOS that sentence hides several different promises, and the storage technology you choose decides which of those promises you can keep. Before we compare frameworks, it helps to name the lifetimes your data can have.

- **Process lifetime.** Values in memory, including `@State` and `@Observable` objects, live until the system terminates the process. On iOS that can happen at any time after your app moves to the background, without warning and without a callback you can rely on.
- **Scene lifetime.** UI state such as the selected tab or a scroll position can be restored when the system recreates a scene. `@SceneStorage` exists for exactly this, and it is deliberately disposable.
- **Install lifetime.** Anything written inside the app's sandbox container survives app launches, device restarts, and app updates, and disappears when the user deletes the app. Most of this book lives here.
- **Backup lifetime.** Files in backed-up directories travel with the user to a new device through an iCloud or computer backup restore. Caches and temporary files do not.
- **Account lifetime.** Data synced through CloudKit lives in the user's iCloud account and appears on every device signed in to it, even after the app is deleted from one of them.
- **Beyond the app.** Keychain items can, in some configurations, outlast the app's deletion on iOS. We treat that as behavior you must not depend on in either direction, and Chapter 5 explains why.

When a product manager says "save the user's settings," your first job is to ask which of these lifetimes they actually want. A sort order probably belongs to the install. A purchase receipt probably belongs to the account. An API token belongs to the device and must never appear in a backup file a user emails to themselves. Choosing the technology follows naturally once the lifetime and the sensitivity are clear.

There is a second axis that matters just as much: the **shape** of the data. A single Boolean flag, a 40-kilobyte JSON document, a 12-megapixel photo, and fifty thousand related records with queries and sorting are all "data," but they stress storage in completely different ways. A technology that is perfect for one shape is painful or dangerous for another. The rest of this chapter combines the two axes, lifetime and shape, into a practical decision guide.

The Kinds of Data in a Real App

Almost every app we have seen in production stores some mix of the following six kinds of data. Learning to sort what you have into these buckets is the single most useful persistence skill, because each bucket has an obvious home.

Preferences

Small values that tune behavior: a sort order, a warning threshold, whether onboarding has been shown. They are read often, written rarely, and losing them is an inconvenience rather than a disaster. They belong in `UserDefaults`, usually through SwiftUI's `@AppStorage`. Chapter 3 covers them in depth.

Documents and exports

Self-contained files the user thinks of as "a thing": an exported backup, a PDF report, an imported spreadsheet. They have a format, often a version number, and sometimes leave the app through the share sheet. They belong in files, usually encoded with `Codable`, written atomically. Chapters 2 and 4 cover them.

Secrets

Passwords, API tokens, encryption keys, and passphrases. They are small, and the cost of leaking them is high. They belong in the Keychain and nowhere else, which is the subject of Chapter 5.

Structured records

The heart of most apps: many records of a few types, connected by relationships, filtered and sorted in different ways on different screens, edited one at a time. This is where a database earns its place. On modern iOS the choices are SwiftData, Core Data, or SQLite directly (possibly through a third-party library). Parts II, III, IV, and Chapter 25 cover them.

Caches and derived data

Anything you can recreate: downloaded images, thumbnails, search indexes, rendered previews. Losing them costs time, not data. They belong in the Caches directory, where the system may purge them under storage pressure and where they are excluded from backups.

Shared and synced data

Data that must be visible to a widget, an App Intent, or another device. Sharing on one device uses an App Group container (Chapter 24); sharing across devices uses CloudKit, either through SwiftData and Core Data mirroring or through your own sync engine (Part V).

> **TIP**
> When you inherit an existing app, list every value it persists and tag each one with a bucket from this section. Mismatches jump out immediately: a token in `UserDefaults`, a thousand records in a JSON file that is rewritten on every keystroke, or thumbnails in `Documents` inflating the user's iCloud backup.

The Decision Table

The table below is the map we return to throughout the book. Each row names a technology, the data it fits best, and the situations where it is the wrong tool. None of these options is "better" in general; each one is excellent for its own shape of data.

Technology	Use it for	Avoid it for
UserDefaults / `@AppStorage`	Small preferences and flags, read synchronously	Secrets, user content, large or growing data
Files + `Codable`	Documents, exports, backups, small whole-file datasets	Data you must query or update piece by piece
Keychain Services	Passwords, tokens, keys, passphrases	Anything large or non-secret
SwiftData (iOS 17+)	Related records with queries, SwiftUI-first apps	Apps that must support iOS 16 or earlier
Core Data	Related records, mature tooling, older OS versions	Teams unwilling to learn its concurrency rules
SQLite directly or GRDB (third-party)	Full SQL control, custom schemas, existing databases	Simple apps that a framework already serves
CloudKit	The same data on every device of one user	Instant or ordered delivery guarantees

Three observations make this table easier to apply. First, most apps use **several** rows at once. Shelfmate uses every single one before the book ends, and that is normal, not a sign of over-engineering. Second, the rows are not mutually exclusive in time: an app can start with a JSON file and graduate to a database when it needs queries, and Chapter 4 shows exactly when that moment arrives. Third, the "avoid" column is where real bugs come from. Developers rarely pick a technology that cannot do the job; they pick one that can do the job badly, and the cost shows up months later as slow launches, lost data, or a security review finding.

A quick taste of each

To make the table concrete, here is roughly what one line of Shelfmate code looks like in each technology. Do not worry about the details; every line is explained in its own chapter.

A one-line taste of each technology (explained in later chapters)

```
// Preferences (Chapter 3)
@AppStorage(SettingsKey.warrantyAlertDays) private var alertDays = 30

// Files + Codable (Chapter 4)
let fileURL = try BackupExporter().write(backup, to: BackupExporter.defaultDirectory)

// Keychain (Chapter 5)
try keychain.set(Data(token.utf8), for: KeychainAccount.insurerToken)

// SwiftData (Chapter 6)
modelContext.insert(Item(name: "Toaster", purchasePriceCents: 4_999))

// Core Data (Chapter 15)
let request = CDItem.fetchRequest()

// SQLite (Chapter 25)
let rows = try database.query("SELECT name FROM item WHERE id = ?",
                              bind: [.integer(8)])
```

Notice that the code gets more ceremonial as you move down. That is the trade-off in miniature: the lower-level the tool, the more control you have and the more decisions you must make yourself.

A decision flow

If you prefer a flow to a table, the questions below reach the same answers. Ask them in order for each piece of data, and stop at the first "yes."

```
Is it a secret (password, token, key)?
  yes -> Keychain
Can it be recreated from somewhere else?
  yes -> Caches directory (or tmp if it is short-lived)
Is it a small preference or flag?
  yes -> UserDefaults / @AppStorage
Is it a self-contained document the user exports or shares?
  yes -> File + Codable, written atomically
Is it many related records you filter, sort, and edit?
  yes -> SwiftData (iOS 17+) or Core Data
         (SQLite directly if you need full SQL control)
Must it appear on the user's other devices?
  add  -> CloudKit sync on top of the choice above
Must a widget or extension read it?
  add  -> App Group container for the choice above
```

The last two questions say "add" rather than "yes" because sync and sharing are not storage technologies in their own right. They are properties you layer on top of a storage choice, and each one brings constraints. CloudKit sync, for example, forbids unique constraints and requires every relationship to be optional. That is why we design Shelfmate's model for CloudKit from the first day, as Chapter 7 explains, even though we only switch sync on in Chapter 22.

SwiftData and Core Data, Honestly

The question we hear most often is "Should I use SwiftData or Core Data?" The honest answer depends on your deployment target, your team, and how much of the framework's surface you need. Here is what we know with confidence.

SwiftData requires iOS 17 or later, and several of its most useful features need iOS 18: compound uniqueness with `#Unique`, indexes with `#Index`, the history API that lets you see what another process changed, and custom data stores. Because Shelfmate targets iOS 18, we can use all of these in the main code path. If you must support iOS 16, SwiftData is simply not available, and Core Data is the framework-level choice.

The two frameworks share technology underneath. SwiftData currently stores its data in an SQLite file through Core Data machinery. Treat that as an implementation detail: Apple has also added a custom data store API, and nothing guarantees the file layout stays the same. What it does mean in practice is that many Core Data lessons transfer directly. Launch arguments such as `-com.apple.CoreData.SQLDebug 1` often help with SwiftData too, the same CloudKit rules apply to both, and Chapter 28 can migrate a Core Data store to SwiftData because the concepts line up.

Core Data is older and more complete. It offers fetched results controllers, batch insert, update, and delete requests, persistent history with fine-grained control, derived attributes, mapping models, staged migration, and sharing with `CKShare` through `NSPersistentCloudKitContainer`. Its tooling, documentation, and community knowledge reflect many years of production use. Its costs are verbosity, a string-based predicate format that fails at run time rather than compile time, and a concurrency model that predates Swift concurrency and needs discipline under Swift 6.

SwiftData is younger and more Swift-native. Models are plain Swift classes with a macro, predicates are type-checked Swift closures, queries plug directly into SwiftUI with `@Query`, and background work maps onto actors through `@ModelActor`. Its costs are a smaller feature set, behavior that has changed between OS releases, and fewer escape hatches when something goes wrong. At the time of writing, for example, SwiftData offers no public-database or sharing API for CloudKit; check the current documentation, because this is exactly the kind of gap that new releases close.

Question	SwiftData	Core Data
Minimum OS	iOS 17 (many features iOS 18)	Every supported iOS version
Model definition	Swift classes with `@Model`	Model editor plus classes

Predicates	Type-checked `#Predicate`	`NSPredicate` format strings
Background work	`@ModelActor`	Contexts and `perform`
CloudKit sharing	Not offered at the time of writing	Yes, with `CKShare`
Batch operations	Batch delete; fewer options	Batch insert, update, delete

Our recommendation for a **new** app targeting iOS 18 is to start with SwiftData unless you already know you need a Core Data-only feature, and to keep your model simple and CloudKit-compatible so that either framework can host it. For an **existing** Core Data app, there is rarely a reason to rewrite working code in a hurry; learn both, and migrate deliberately when a concrete benefit justifies it. Chapter 28 shows three ways to do that safely.

> **NOTE**
>
> Third-party options exist too. **GRDB.swift** and **SQLite.swift** are popular open-source SQLite libraries, and Realm is another database you may meet in older projects; check its current maintenance status before adopting it. This book treats third-party libraries as alternatives worth knowing about, and Chapter 25 explains when SQLite-level control is worth the extra work.

Meet Shelfmate

Learning persistence from disconnected snippets is frustrating, because the hard parts only appear when real features interact: a delete that must also remove a file, a widget that reads the store while the app writes to it, a migration that has to keep old data meaningful. So we build one app throughout the book.

Shelfmate is a fictional home inventory app. You record what you own, room by room, with purchase dates, prices, serial numbers, warranties, photos, and tags. If a pipe bursts or a burglar visits, Shelfmate tells you what you had and what it cost. If a warranty is about to expire, Shelfmate warns you before it does. The domain is ordinary on purpose: everyone understands it, yet it exercises every persistence technique in this book.

Features and where they live

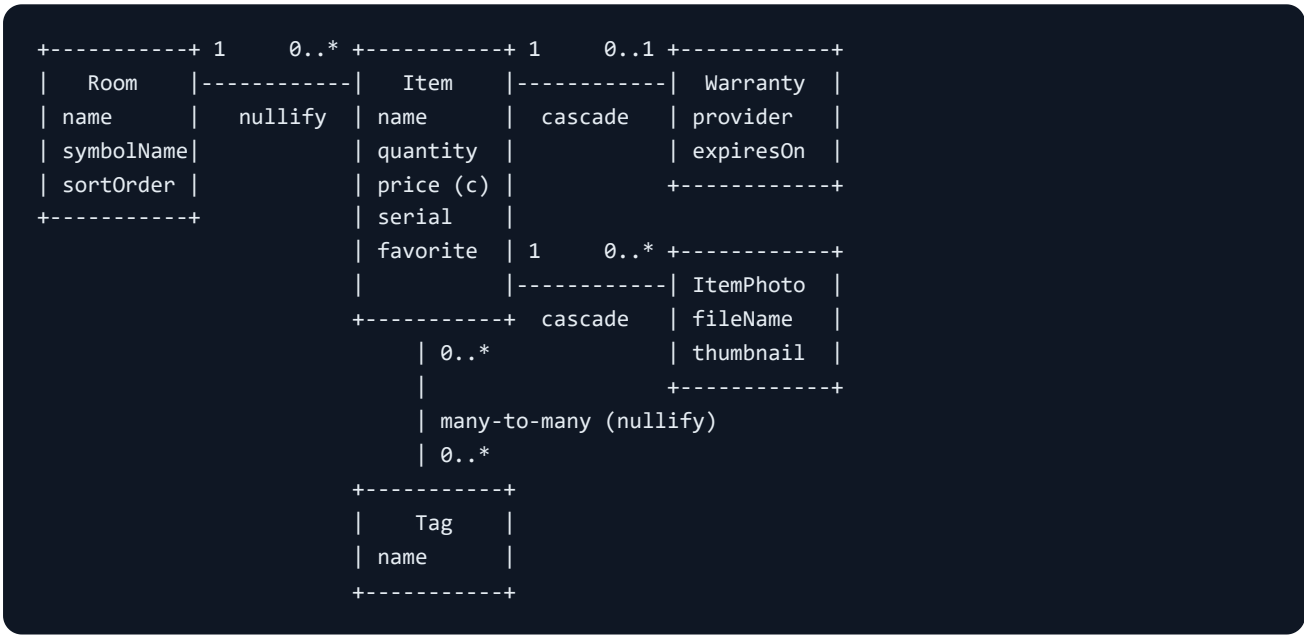
Feature	Storage technique	Chapter
Rooms, items, warranties, tags	SwiftData models and relationships	6–9
Item photos	Files plus thumbnails with external storage	10
CSV and backup import	Background `@ModelActor`	11
Upgrading old stores	Versioned schemas and migration plans	12
Settings	`UserDefaults` and `@AppStorage`	3

JSON backups	Files, `Codable`, atomic writes	4
Insurer API token	Keychain	5
iCloud sync	SwiftData with CloudKit	22
Warranty widget	App Group container and shared store	24

The app has four tabs: **Rooms** lists rooms with item counts and total value; **Items** lists every item with search and a sort order; **Warranties** groups warranties into Expired, Expiring Soon, and Active sections; and **Settings** holds preferences, backups, and the insurer connection. The insurer is a fictional company called Coverwell Insurance, whose pretend API lives at `https://api.coverwell.example/v1/`. We never call it for real; it exists so Chapter 5 has a realistic secret to protect.

The data model at a glance

Shelfmate's model has five types. Chapter 7 designs them property by property; for now, the sketch below shows how they connect.



The labels on the lines are delete rules, and they encode real product decisions. Deleting a room does not delete its items; they become "Unassigned," because moving out of a room should not erase your sofa. Deleting an item deletes its warranty and its photos, because neither means anything on its own. Deleting a tag simply removes it from the items that carried it. Chapter 7 shows how each rule is declared and what happens in the store when it fires.

Two further design decisions run through the whole book. First, **money is stored as an integer number of cents** in a property called `purchasePriceCents`, never as a floating-point dollar amount. Binary floating point cannot represent most decimal fractions exactly: in Swift, as in any language that uses IEEE-754 doubles, `0.1 + 0.2` is `0.30000000000000004`. Those tiny errors become visible pennies

when you sum and round, so we avoid them at the source. Second, **every property has a default value or is optional, and every relationship is optional**. That is a CloudKit requirement, and designing for it on day one costs nothing while retrofitting it later costs a migration.

The sample data

Every chapter works with the same sample household, so you can check your results against ours. `SampleData.insert(into:)` inserts it into a SwiftData context, and the Core Data and raw SQLite chapters load identical rows. The household has five rooms and fifteen items.

Room	Items	Value	Examples
Kitchen	4	$830.49	Espresso Machine, Stand Mixer, six Wine Glasses
Living Room	3	$2,287.00	Sofa, 4K Television, Floor Lamp
Home Office	3	$2,397.99	Laptop, Office Chair, Desk Monitor
Bedroom	2	$1,298.00	Mattress, Air Purifier
Garage	2	$908.00	Cordless Drill, Bicycle
(Unassigned)	1	$49.98	Two Travel Adapters

Across all rooms that is **15 items and 21 units** (the Wine Glass row has a quantity of 6 and the Travel Adapter row a quantity of 2), **10 warranties**, **9 tags**, and **4 favorites**: the Espresso Machine, the 4K Television, the Laptop, and the Bicycle. The total inventory value is **777,146 cents, or $7,771.46**, computed as quantity times unit price for each item.

Dates are the other half of the sample. Warranty logic depends on "today," and a book cannot know when you read it, so every calculation uses a fixed **reference date of 2026-03-15**, available in code as `SampleData.referenceDate`. As of that date the Office Chair's warranty expired 15 days ago, the 4K Television's expires in 16 days, and the Stand Mixer's in 18 days. With the default 30-day alert window, those three are exactly the warranties that need attention, and they are exactly what the widget in Chapter 24 shows.

Shelfmate/Persistence/SampleData.swift (dates; Chapter 6 adds insert(into:))

```
import Foundation

enum SampleData {
    /// Noon local time on the given day, Gregorian calendar.
    /// Noon keeps the calendar day stable when the time zone changes.
    static func day(_ year: Int, _ month: Int, _ day: Int) -> Date {
        let calendar = Calendar(identifier: .gregorian)
        let components = DateComponents(year: year, month: month,
                                        day: day, hour: 12)
        // The sample dates are all valid, so this cannot fail.
        guard let date = calendar.date(from: components) else {
            preconditionFailure("Invalid sample date \(year)-\(month)-\(day)")
        }
```

```
        return date
    }

    /// The "today" used by every calculation in the book: 2026-03-15.
    static let referenceDate = day(2026, 3, 15)
}
```

Why noon? A date stored at midnight can slide into the previous or next calendar day when the device changes time zone or a daylight saving transition shifts the clock. Noon leaves twelve hours of margin in each direction, so "purchased on 2025-11-28" stays on the 28th. It is a small habit that removes a whole category of off-by-one bugs from tests.

> **WARNING**
>
> Never call `Date.now` inside logic you want to test. Pass the date in, as `Warranty.daysRemaining(asOf:)` and `Warranty.needingAttention(asOf:)` do, and let the view supply `Date.now` while tests supply `SampleData.referenceDate`. Code that reads the clock directly produces tests that pass today and fail next month.

Shelfmate and ShelfmateCD

The main project, **Shelfmate**, uses SwiftData. Part IV teaches Core Data, and teaching it on a different domain would waste the familiarity you have built by then. So Part IV uses a parallel teaching project, **ShelfmateCD**, with the same five entities, the same relationships and delete rules, and the same fifteen sample items. Its Core Data classes carry a `CD` prefix (`CDRoom`, `CDItem`, `CDWarranty`, `CDItemPhoto`, `CDTag`) so you can always tell at a glance which framework a line belongs to. Its store file is `ShelfmateCD.sqlite`, managed by a `PersistenceController` type.

Because the data is identical, the numbers are identical too. When Chapter 17 sums value per room with an `NSExpression`, you should see the same $830.49 for the Kitchen that Chapter 8 computes with SwiftData and that Chapter 25 computes with raw SQL. Matching numbers across three technologies is a surprisingly strong check that you understand each of them. Finally, Chapter 28 migrates ShelfmateCD's store into SwiftData and verifies the result with the same totals: 15 items and $7,771.46.

The Project Layout

Every code listing in this book carries a caption with its file path, so you always know where a piece of code lives. The layout below is the finished project; early chapters create only some of these files, and later chapters add the rest.

```
Shelfmate/
  ShelfmateApp.swift              @main, creates the ModelContainer
  ContentView.swift               TabView: Rooms, Items, Warranties, Settings
  Model/
```

```
    Room.swift  Item.swift  Warranty.swift  ItemPhoto.swift  Tag.swift
    Item+Computed.swift             totals, warranty status
    Queries.swift                   predicates and fetch descriptors
    Money.swift                     cents formatting and parsing
    Schema/                         versioned schemas (Chapter 12)
  Persistence/
    ShelfmateContainer.swift        container factory
    SampleData.swift                the sample household
    BackupFile.swift  BackupExporter.swift  InventoryFileStore.swift
    KeychainStore.swift  PhotoStore.swift  ThumbnailMaker.swift
    ItemImporter.swift  HistoryMonitor.swift
    SyncEngine/ShelfmateSyncEngine.swift
  Settings/
    SettingsKey.swift  SettingsView.swift
  Shared/
    AppGroup.swift                  compiled into app and widget
  Views/                            lists, detail, editor, photo grid
  Resources/shelfmate-import.csv
  PrivacyInfo.xcprivacy
ShelfmateWidget/                    warranty widget and App Intent
ShelfmateTests/                     model, query, importer, migration,
                                    backup tests
ShelfmateCD/                        the Core Data edition (Part IV)
```

The split between `Model/` and `Persistence/` is deliberate. `Model/` holds the types that describe the domain and the questions you ask of it; it knows nothing about where files live. `Persistence/` holds everything that touches the disk, the Keychain, or the network: containers, importers, exporters, and file stores. Views depend on both, but the persistence types never depend on views. That one-way dependency is what lets Chapter 27 test persistence code without launching any UI.

The identifiers are fixed too. The bundle identifier is `com.example.shelfmate`, the widget's is `com.example.shelfmate.widget`, the App Group is `group.com.example.shelfmate`, and the iCloud container is `iCloud.com.example.shelfmate`. Replace `com.example` with your own reverse domain when you build the project; capabilities such as App Groups and iCloud require identifiers registered to your developer team.

Conventions Used in This Book

Swift 6 and strict concurrency

All code is written for the **Swift 6 language mode**, which turns data-race checking from warnings into errors. In practice that shapes our code in visible ways. Types that cross actor boundaries are `Sendable`: our data transfer objects, settings keys, and helper structs all are. UI-touching code is on the main actor; SwiftUI views already are, and helpers such as `SampleData.insert(into:)` are marked `@MainActor` because they use the main context. Model objects from SwiftData and managed objects from Core Data are **not** `Sendable`, so we never pass them between actors; we pass identifiers or plain values instead. Chapter 11 and Chapter 16 explain the rules in depth, and you will see the same pattern on nearly every page.

Recent versions of Xcode can also configure new projects to isolate code to the main actor by default. The code in this book does not rely on that setting: we annotate isolation explicitly, so it reads the same either way. If your project uses main-actor-by-default, a few of our annotations become redundant but harmless.

Deployment target: iOS 18

Shelfmate's deployment target is **iOS 18**. Everything in the main code path works there. SwiftData basics arrived in iOS 17, but we rely on several iOS 18 additions, and the first time each one appears we label it "iOS 18+" so you know what to adjust if your own target is lower.

iOS 26 side notes

Where iOS 26 adds something worth knowing, we put it in a clearly marked note that begins "iOS 26:" and guard any code with `if #available(iOS 26, *)` or `@available(iOS 26, *)`. Those notes are deliberately brief and hedged, because Apple's documentation is the authority on the newest releases. Nothing in the main path depends on them.

Output you should see

When we show console output, we introduce it as "you should see something like." Framework descriptions of objects, error message wording, and file paths inside the simulator vary by OS version and contain random identifiers, so we show their shape rather than promising every character. Paths are abbreviated with `.../Library/Application Support/...` where the full path would contain a device-specific UUID.

Output (you should see something like)

```
Store: .../Library/Application Support/Shelfmate.store
Rooms: 5  Items: 15  Units: 21
Total value: $7,771.46
```

Numbers you can trust

Every number in the sample results, from day counts to cents, has been computed independently rather than estimated, and the same totals appear wherever the same question is asked. What we do **not** give you are performance numbers: no "ten times faster," no milliseconds, no storage quotas. Those depend on the device, the OS version, and your data, and the only trustworthy measurement is the one you take yourself with Instruments. Where performance matters, we explain why one approach is typically faster and show you how to measure it.