

PROJECT COURSE

Personal Catalogue: iOS (Swift & SwiftUI)

10 Chapters

Philip Osztromok

Generated with Claude

PERSONAL CATALOGUE: IOS (SWIFT & SWIFTUI)

Contents

Chapter 1	Project Setup	2
Chapter 2	Data Model	10
Chapter 3	Persistence Layer	19
Chapter 4	The List Screen	30
Chapter 5	The Detail Screen	39
Chapter 6	Add and Edit Form	47
Chapter 7	Tags	57
Chapter 8	Search and Filter	64
Chapter 9	Polish and Errors	72
Chapter 10	Backup, Tests and Next Steps	80

Personal Catalogue: iOS (Swift & SwiftUI)

Course 1 · Chapter 1 · The App and the Empty Project

This course builds a real app for your own iPhone: a catalogue of the books, CDs, DVDs and Blu-rays you own, so that standing in a shop you can answer "*do I already have this?*" in a few seconds. You may already have built it for the web with PHP, Django, Express or Firebase. This is the same app, rebuilt as a native iOS app in **Swift** with **SwiftUI** for the screens and **SwiftData** for the database. A sister course, *Personal Catalogue: Android (Kotlin & Compose)*, builds exactly the same app with the same chapter order, so you can follow either one, or both side by side.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Nothing in this chapter was run.** The machine this course is written on is a Windows PC with no Mac, no Xcode and no simulator, and Xcode only runs on macOS. The steps and code come from my knowledge of Xcode and SwiftUI and from the search results listed at the end of the chapter. **Xcode's wizard wording, its default deployment target and its generated files change with every release**, so wherever this chapter names one, trust what your own Xcode shows over the text here, and treat the code as a starting point to compile, not as something already proven to build.

What We Are Building

The app has one job and a short list of features, taken from the shared spec that both mobile courses use:

CHAPTER	FEATURE ADDED
1	The empty project, folder layout, first screen in the simulator
2–3	The data model and the SwiftData store, seeded with the four item types
4–5	The list screen and the detail screen, with navigation between them
6	One add/edit form whose labels change with the item type
7	Tags: a managed, reusable list, with suggestions as you type
8	Search by title or creator, filter by tag
9	Duplicate warning, dark appearance, accessibility, error states

CHAPTER	FEATURE ADDED
10	JSON backup and restore, tests, and what to build next

The data model is small. An **item** has a title, a creator (author, artist or director), a "format detail" (hardcover, album, region code and so on), an optional year and some notes. Every item has one **type** (Book, CD, DVD or Blu-ray) and any number of **tags**. The app is for one person, stores everything on the phone and works with no network. Barcode scanning, cover art and cloud sync are deliberately *not* in this course: they are listed in the spec as later features.

How This Differs from the Web Versions

If you built a web catalogue, most of the thinking carries over and only the machinery changes:

	WEB CATALOGUE (DJANGO, PHP, EXPRESS)	THIS COURSE
Language	Python, PHP or JavaScript	Swift
Screens	HTML templates or React components	SwiftUI `View` structs
Database	PostgreSQL, MySQL, MongoDB or Firestore on a server	SwiftData, *on the phone*
Data entry	Django admin or an HTML form	A SwiftUI `Form` screen
Server	Needed (Gunicorn, Apache, Node and so on)	None: the app is the whole system
Running it	A browser pointed at a URL	The iOS Simulator (on a Mac), or your own iPhone
Updates	Deploy and everyone sees them	Install a new build; data lives on that one phone

The big consequence is the last two rows. There is no server and no login, so the app is simple to run but the data lives on one device. That is why Chapter 10 builds a backup file: it is the iOS equivalent of the database dump you would take from a web version.

What You Need

A Mac with Xcode. iOS apps can only be built with Xcode, which only runs on macOS. *iOS Development: Fundamentals* and *Xcode: The IDE Itself* cover installing and using it, so this chapter does not repeat that. One search source states that SwiftData needs Xcode 15.0 or later and an iOS 17.0 or later deployment target, and that apps that must support earlier versions have to use Core Data instead. That is why this course uses SwiftData with a modern minimum version.

Creating the Project

1. In Xcode choose **File > New > Project** (or **Create New Project** on the welcome screen).
2. Choose the **iOS** tab and the **App** template, then Next.
3. Set **Product Name** to `Catalogue`. Pick your **Team** (a free personal team is enough to run on a simulator or your own phone) and an **Organization Identifier**, such as `com.example`; Xcode combines them into the bundle identifier.
4. Set **Interface** to **SwiftUI** and **Language** to **Swift**.
5. If the dialog offers a **Storage** option, choose **None** for now. One of the search sources recommends picking SwiftData in the wizard, which makes Xcode add a starter `Item` model and some container code; we write our own model in Chapter 2, so we start without it. If your Xcode has no such option, or you chose SwiftData anyway, just delete the generated `Item.swift` later when Chapter 2 replaces it.
6. Choose where to save it and click **Create**.
7. Select the project in the navigator, then the **Catalogue** target, and look at **Minimum Deployments**. A forum post reports that on Xcode 16.2 the template's default was iOS 18.2, well above 17.0, so check what yours says and lower it only if you need to support older phones. Do not set it below 17.0 for this course.

What the Wizard Generated

```
Catalogue/
├── Catalogue.xcodeproj          <- the project file (build settings, targets)
├── Catalogue/
│   ├── CatalogueApp.swift       <- the entry point (@main)
│   ├── ContentView.swift        <- the first screen
│   └── Assets.xcassets          <- app icon, colours, images
```

```
├── CatalogueTests/             <- unit tests (Chapter 10)
└── CatalogueUITests/           <- UI tests
```

`CatalogueApp.swift` is where the app starts: a struct marked `@main` that conforms to `App` and returns a `WindowGroup` containing the first view. `ContentView.swift` holds that first view. The exact file list and the names of the test folders can differ in your Xcode.

A Folder Layout for the Whole Course

The wizard puts everything in one group. For a small app that is fine, but this one grows through ten chapters, so decide the layout now. The groups are by *responsibility*, the same split the web versions made between models, views and templates:

```
Catalogue/
├── CatalogueApp.swift
├── Models/             <- ItemType, Item, Tag as SwiftData models (Chapters 2-3)
├── Data/               <- seeding and the small repository (Chapter 3)
├── Views/
│   ├── List/           <- list screen (Chapter 4)
│   ├── Detail/         <- detail screen (Chapter 5)
│   └── Form/           <- add/edit screen (Chapter 6)
├── Backup/             <- JSON export and import (Chapter 10)
└── Assets.xcassets
```

In the Project navigator, right-click the `Catalogue` group and choose **New Group** for each of `Models`, `Data`, `Views` (with `List`, `Detail` and `Form` inside) and `Backup`. Groups are for organising the navigator, and may or may not match folders on disk, depending on how your Xcode created the project; either way the layout works. Creating them now is optional: the point is to know where each thing will go.

A First Screen of Our Own

Replace the generated "Hello, world" with a screen that says what the app is. This is a sketch of the usual shape of the two generated files; yours may include a preview macro or other lines, which you should keep.

```swift
// CatalogueApp.swift
import SwiftUI

@main
struct CatalogueApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}

// ContentView.swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        NavigationStack {
            Text("My Catalogue")
                .font(.largeTitle)
                .navigationTitle("Catalogue")
        }
    }
}

#Preview {
    ContentView()
}
```

Three ideas in that code carry the whole course. A **View** is a Swift struct whose `body` *describes* some UI; SwiftUI calls `body` again whenever the data it shows changes. A **NavigationStack** is the standard container for screens that push one another (the list will push the detail screen in Chapter 5), and it supplies the title bar. A **modifier** such as `.font(.largeTitle)` is a method that wraps a view in a changed copy; you chain them, and the order matters.

Running It

1. Choose a simulated iPhone from the device menu at the top of the Xcode window.
2. Press **Run** (the play button, or Command-R). The first build and simulator start-up are slow; later ones are much faster.

3. When the simulator shows *My Catalogue*, you have a working toolchain. The canvas preview (the `#Preview` block) shows the same screen without running the app.

To run on your own iPhone, connect it by cable, trust the computer on the phone, turn on Developer Mode in the phone's settings, and choose it in the device menu; Xcode signs the build with your personal team. If the build fails, the Issue navigator lists the errors; fix the first one before the rest, since later errors often cascade from it.

> **COMPARE WITH THE ANDROID COURSE**
>
> In *Personal Catalogue: Android (Kotlin & Compose)* Chapter 1 does the same job: a new Android Studio project, a package layout, and a first screen. The mapping is direct: a SwiftUI `View` is a composable function, a `NavigationStack` with a title is roughly a `Scaffold` with a top bar, and the iOS Simulator is the Android Emulator. The two apps end up with the same screens and the same data model.

Challenges

EXERCISE 1

Create the project as described and run it in the simulator. Write down the **Product Name**, **Bundle Identifier**, **Minimum Deployment** and the options your Xcode offered for **Storage**. Then explain in one sentence why this course needs a deployment target of iOS 17 or later.

View solution

EXERCISE 2

Change the screen so that it shows the app name and, underneath it, a smaller line: "*Books, CDs, DVDs and Blu-rays*". You will need a `VStack`. Run it and say which argument controls the gap between the two lines.

View solution

EXERCISE 3

Open the web catalogue course you finished (any of the four) and list, for each of the seven rows in the comparison table in this chapter, the exact file or tool you used there. Then pick the one row where iOS makes your life easier and the one where it makes it harder, and explain each in a sentence.

View solution

CHAPTER 1 QUICK REFERENCE

- The app: single-user, offline catalogue of books, CDs, DVDs and Blu-rays; items have a type, a creator, a format detail, an optional year, notes and tags
- Stack: Swift, SwiftUI, SwiftData, `@Observable` view models, `NavigationStack`
- SwiftData needs Xcode 15+ and an iOS 17+ deployment target (one source; apps for earlier versions use Core Data)
- New project: File > New > Project > iOS > App; Interface SwiftUI; Language Swift; Storage None for now
- Folder layout: `Models`, `Data`, `Views/List`, `Views/Detail`, `Views/Form`, `Backup`
- A `View` is a struct with a `body`; `NavigationStack` holds pushed screens; modifiers chain and order matters
- Run: choose a simulator, press Run (Command-R); or use your own iPhone with Developer Mode on

NOT YET VERIFIED

Checked (one web search on 2026-11-19, results mostly forums and blogs, not Apple's own documentation): that the SwiftUI, Swift, SwiftData combination is the usual wizard choice; that SwiftData needs Xcode 15.0 or later and an iOS 17.0 or later deployment target; that choosing SwiftData in the wizard adds a starter `Item` model; and one developer's report that the default deployment target on Xcode 16.2 was iOS 18.2.
Not checked, from general knowledge: the wizard's exact field names and whether your Xcode offers a Storage option with a None choice; the generated file list; the code in both files (including `#Preview` and `NavigationStack` and that they compile together); the group layout (a design choice, not a platform rule); the simulator and Developer Mode steps; free personal team signing; and the comparison with the Android course. Nothing was built or run. Compile it, and correct anything your Xcode does differently.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Product Builder, SwiftUI iOS app; Mehmet Baykar, SwiftData basics; Apple Developer Forums, Choosing minimum deployment targets; Andrew Bancroft, Add SwiftData to an existing app. Only the titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: iOS (Swift & SwiftUI) 2 designs the data model: the `ItemType`, `Item` and `Tag` classes as SwiftData models, the many-to-many link between items and tags, and why the same decisions you made for the Django models apply here.

Personal Catalogue: iOS (Swift & SwiftUI)

Course 1 · Chapter 2 · @Model, Relationships and Delete Rules

Chapter 1 left you with an empty project and a first screen. This chapter writes the **shape of the data**: three Swift classes that SwiftData will store. There is no container, no query and no screen yet; those come in Chapters 3 and 4. As in the Android course, the model comes first because every later chapter reads or writes it, and changing a stored model after the app is installed means a migration.

> **WHAT WAS RUN, AND WHAT WAS NOT, AND AN HONEST WARNING ABOUT SOURCES**
>
> **Nothing in this chapter was compiled or run**: there is no Mac or Xcode on this machine. More than usual, **I could not check this chapter against Apple's own reference pages**: my two searches returned the WWDC23 session "Model your schema with SwiftData", Apple developer-forum threads, and third-party articles, and those **disagree with each other** on how many-to-many relationships declare their inverse. I chose the pattern the sources most agree is safe, and the Not Yet Verified box lists exactly what rests on community reports. **Build this chapter early, and trust Xcode's errors over my text.**

From Room to SwiftData

If you have read the Android course's Chapter 2, here is the translation. SwiftData needs less code, because it keeps the join between items and tags to itself.

CONCEPT	DJANGO	ANDROID (ROOM)	IOS (SWIFTDATA)
A table	`models.Model`	`@Entity data class`	`@Model final class`
Primary key	Automatic `id`	`@PrimaryKey Long`	None to write: every model has a built-in `persistentModelID`
Optional field	`null=True`	`Int?`	`Int?`
Unique value	`unique=True`	`Index(unique = true)`	`@Attribute(.unique)`
One-to-many	`ForeignKey`	An id column plus `@ForeignKey`	A property of the other model's type, e.g. `var type: ItemType?`

CONCEPT	DJANGO	ANDROID (ROOM)	IOS (SWIFTDATA)
Many-to-many	`ManyToManyField`	An explicit cross-reference entity	An array on each side, `var tags: [Tag]`; the join is automatic
On delete	`on_delete=`	`onDelete = CASCADE` / `RESTRICT`	`@Relationship(deleteRule:)`
Creating tables	`migrate`	Generated from entities by the database class	Generated from the `@Model` classes by the container (Chapter 3)

The Three Models

SwiftData models are **classes** (not structs) marked with the `@Model` macro, which makes them observable by SwiftUI and storable. Put each in its own file in a *Models* group in Xcode (right-click the `Catalogue` folder, **New Group**, then **New File > Swift File**). If your wizard generated an `Item.swift`, replace its contents: its starter `Item` class has the same name as ours.

ItemType

The fixed lookup list (Book, CD, DVD, Blu-ray) that Chapter 3 will seed. Its name is unique. It also lists its items, and its delete rule is `.deny`, which makes SwiftData refuse to delete a type that still has items, the same protection as `RESTRICT` on Android.

```
import Foundation
import SwiftData

@Model
final class ItemType {
    @Attribute(.unique) var name: String

    @Relationship(deleteRule: .deny, inverse: \Item.type)
    var items: [Item] = []

    init(name: String) {
        self.name = name
    }
}
```

Tag

A tag is a name, and the spec says "Web Development" and "web development" must be one tag. SwiftData has no equivalent of Room's case-insensitive column setting, so the standard answer is to store two things: the **display name** the user typed (kept as first typed, so the screen shows "Sci-Fi") and a **key**, a folded version of it, which is the property marked unique. Two names that fold to the same key are the same tag.

```
import Foundation
import SwiftData

@Model
final class Tag {
    /// What the user typed, cleaned up: "Sci-Fi".
    var name: String

    /// The folded form used for uniqueness: "sci-fi". Never shown.
    @Attribute(.unique) var key: String

    /// The other side of Item.tags. Declared plainly; the inverse is declared on Item.
    var items: [Item] = []

    init(name: String) {
        let cleaned = Tag.cleaned(name)
        self.name = cleaned
        self.key = Tag.key(for: cleaned)
    }

    /// Trims the name and turns any run of whitespace into one space.
    static func cleaned(_ raw: String) -> String {
        raw.split(whereSeparator: \.isWhitespace).joined(separator: " ")
    }

    /// The uniqueness key: cleaned, then folded for case and accents.
    static func key(for raw: String) -> String {
        cleaned(raw).folding(options: [.caseInsensitive, .diacriticInsensitive], locale:
    }
}
```

Two things to notice. **Foundation's folding is Unicode-aware**: "Café" and "cafe" fold to the same key, which the Android version's ASCII-only `NOCASE` could not do. Whether that is what you *want* is a design choice (Exercise 3 asks you to decide), but it is available. And the `key`

must always be produced from the name by `Tag.key(for:)`, never typed in by hand, or the two can drift apart.

Item

The main model. It holds a to-one relationship to its type and a many-to-many relationship to its tags.

```
import Foundation
import SwiftData

@Model
final class Item {
    var title: String
    var creator: String
    var formatDetail: String
    var releaseYear: Int?
    var notes: String
    var createdAt: Date

    var type: ItemType?

    @Relationship(inverse: \Tag.items)
    var tags: [Tag] = []

    init(
        title: String,
        creator: String = "",
        formatDetail: String = "",
        releaseYear: Int? = nil,
        notes: String = "",
        createdAt: Date = .now,
        type: ItemType?
    ) {
        self.title = title
        self.creator = creator
        self.formatDetail = formatDetail
        self.releaseYear = releaseYear
        self.notes = notes
        self.createdAt = createdAt
        self.type = type
    }
}
```

Three decisions. `type` is *optional* because SwiftData's to-one relationships are, so "every item has a type" becomes a rule your own code (the form, in Chapter 6) must enforce rather than something the database does. `createdAt` is a `Date`, which SwiftData stores directly (Room needed a `Long`). And there is **no id**: each model instance already has a `persistentModelID`, which Chapter 5 will use to say *which* item to show.

The Many-to-Many, and Why Only One Side Declares the Inverse

The tag relationship is the part of this chapter you should read twice, because it is where the sources disagree. The facts, as best I could establish them:

- A many-to-many relationship in SwiftData is just an **array on each side**: `Item.tags` and `Tag.items`. SwiftData creates the join storage; you never see a link table (unlike Room's cross-reference entity).
- Some sources say SwiftData infers the inverse when at least one end is optional; others say it never infers a many-to-many inverse and you must declare it; another reports that for many-to-many the inverse can only be declared on *one* side. Apple's own forum has a many-to-many question that ends in a "Failed to fulfill link" error.
- The pattern all of those allow is the one used above: `@Relationship(inverse: \Tag.items)` on `Item.tags` only, both properties arrays with a default of `[]`.

In Chapter 3 you will build a container and insert real data; **that is the first time the relationship is actually exercised**, so if it misbehaves, the first thing to try is moving the inverse declaration to the other side or declaring it explicitly on both, and re-reading Apple's current `Relationship` documentation, which I could not.

Delete Rules in This Model

SwiftData's `@Relationship(deleteRule:)` decides what happens to the *other* side when an object is deleted. The WWDC23 session and forum posts agree that the default is **nullify**: the relationship is simply cleared. That default is exactly what we want for tags, and the explicit `.deny` is what we want for types.

YOU DELETE	RULE	EFFECT	ANDROID EQUIVALENT
An item	Default (nullify)	It disappears from every tag's `items`; the tags stay	`CASCADE` on the link rows
A tag	Default (nullify)	It disappears from every item's `tags`; the items stay	`CASCADE` on the link rows
An item type that has items	`.deny`	The delete is refused	`RESTRICT`

Be careful with the one rule you might be tempted to add: `.cascade` on `Tag.items` would delete every item carrying a deleted tag, which is almost never wanted. Several forum threads also report cascade rules failing in some situations (one links it to declaring an inverse; another to saving the context explicitly), so cascade is a rule to test, not to assume. We have no need for it.

Helper Properties

Screens will constantly want a tag list in alphabetical order and a type name that is not optional. Put those once on the model, as computed properties, and add them to the end of `Item`:

```
extension Item {
    /// "Book", "CD"... or an empty string if (against the rules) there is no type.
    var typeName: String { type?.name ?? "" }

    /// The item's tags in alphabetical order, ignoring case.
    var sortedTags: [Tag] {
        tags.sorted { $0.name.localizedCaseInsensitiveCompare($1.name) == .orderedAscend
    }
}
```

Computed properties are not stored, so adding or changing them never needs a migration; only changes to stored properties do.

WHY MODELS, NOT AN ENUM, FOR ITEMTYPE

Swift makes an `enum ItemType: String, Codable` very easy, and for a list that never changes it would be simpler, with no seeding, no relationship and no delete rule. This course keeps `ItemType` a model for two reasons: the spec lists it as a table so that the two mobile courses mirror each other, and a model is the version that survives the roadmap, where a user might add their own types (vinyl, board games) without a new app release. If you prefer the enum, everything in Chapters 3 to 10 gets shorter, and that is a perfectly good variation to try.

COMPARE WITH THE ANDROID COURSE

Personal Catalogue: Android (Kotlin & Compose) Chapter 2 builds the same data with four classes instead of three: its `ItemTagCrossRef` is the join that SwiftData hides. Android makes the `id`, the foreign key, the indices and the unique rules explicit and checks them when the project builds; SwiftData infers most of them from the properties and relationships. The cost of that brevity is the one this chapter has been warning about: more of the behaviour is implicit, so more of it has to be tested by running the app.

Challenges

EXERCISE 1

Create the three model files and the helper extension, delete any wizard-generated `Item.swift` that conflicts, and build the project (**Product > Build**). Write down every error or warning and which line it pointed at. If the build succeeds, note that a successful build does *not* yet prove the relationship works; say why.

View solution

EXERCISE 2

Sketch the stored objects after you add a book "Dune" tagged *sci-fi* and *classic*, and a DVD "Alien" tagged *sci-fi*. Say what each object's relationship properties contain. Then say what changes if you delete the tag *sci-fi*, what changes if you delete "Dune", and what should happen if you try to delete the type DVD.

View solution

EXERCISE 3

Work out by hand what `Tag.cleaned` and `Tag.key(for:)` return for: `" Sci-Fi "`, `"sci fi"`, `"SCI-FI"`, `"Café"` and `"cafe"`. Decide whether "Café" and "cafe" should be the same tag, and say what you would change in `key(for:)` if not. Finally, explain what a unique key means for the line `context.insert(Tag(name: "SCI-FI"))` when "Sci-Fi" already exists.

View solution

CHAPTER 2 QUICK REFERENCE

- Model: `@Model final class X { ... }` with a designated `init`; no id to write (each has `persistentModelID`)
- Optional field: `var releaseYear: Int?`; unique value: `@Attribute(.unique)`
- To-one relationship: `var type: ItemType?` (always optional); to-many: `var items: [Item] = []`
- Many-to-many: an array on each side; declare `@Relationship(inverse: \Tag.items)` on *one* side only (the safest pattern across sources: test it)
- Delete rules: default nullify (clears the link); `.deny` refuses deletion while related objects exist; avoid `.cascade` on tags
- Case-insensitive unique names: store a folded `key` (`folding(options: [.caseInsensitive, .diacriticInsensitive], locale: nil)`) and mark it unique
- Computed properties (`typeName`, `sortedTags`) are free: no migration

NOT YET VERIFIED

Checked (two web searches on 2026-11-19), and only through secondary sources: that the WWDC23 session and forum posts describe nullify as the default delete rule, with cascade available by adding `@Relationship`; that third-party notes on the WWDC23 session say `@Attribute(.unique)` makes an insert update an existing record with the same value; that sources conflict about inferring many-to-many inverses (one says inferred when an end is optional, one says never inferred, one says the inverse may only be set on one side); that Apple's forum has a many-to-many "Failed to fulfill link" error report; and that some forum posts report cascade failing in certain setups.
Not checked at all, from general knowledge: that `.deny` exists as a delete rule and behaves as described (it was not in the search results); that the `@Relationship(inverse: \Tag.items)` key-path syntax is right for your SwiftData version; the Foundation `folding(options:locale:)` behaviour on your strings; that a unique `key` plus a plain `name` is how to get case-insensitive uniqueness (my design, not a documented recipe); that `final class` is required; the minimum OS version; and all

code, **none of which was compiled. I did not consult Apple's documentation pages for `Relationship`, `Attribute` or `Model`: please read them.**

SOURCES FOR THIS CHAPTER

Search results (2026-11-19), none of them Apple reference pages: WWDC23: Model your schema with SwiftData; WWDC Notes on that session (third party); Apple Developer Forums: SwiftData many-to-many "PendingRelationshipLink"; Apple Developer Forums: SwiftData does not cascade delete; Fatbobman: Relationships in SwiftData (third party); SwiftData: Relationships (third-party tutorial). Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: iOS (Swift & SwiftUI) 3 builds the persistence layer: the model container that creates the store, seeding the four item types on first launch, a small repository for the rules that are more than one line, and SwiftUI previews that run on an in-memory store.

Personal Catalogue: iOS (Swift & SwiftUI)

Course 1 · Chapter 3 · Container, Seeding, Repository and Previews

Chapter 2 described the data; this chapter makes it real. You will create the **model container** that opens the SwiftData store, seed the four item types on first launch, write a small **repository** for the rules that are more than one line (saving an item with its tags, finding or creating a tag), and set up **previews** that run on an in-memory store so that SwiftUI's canvas never touches your real data. By the end the app can store and load a catalogue, though nothing on screen uses it until Chapter 4.

> **WHAT WAS RUN, AND WHAT WAS NOT, AND WHAT THE SOURCES ARE**
>
> **Nothing in this chapter was compiled or run**: there is no Mac or Xcode on this machine. My search for this chapter returned tutorials (Kodeco, AppCoda, Hacking with Swift forum posts) and Apple developer-forum threads, **not Apple's reference pages**, and found **nothing on** `ModelContext.save()`, `fetchCount` **or transactions**, so those are from my memory. The Not Yet Verified box separates what the sources showed from what I assumed. As in Chapter 2: build early, and trust Xcode's errors over my text.

What Is Different from Android Here

Room needed a lot of machinery in this layer: DAO interfaces, a database class, a Hilt module. SwiftData has far less, because a SwiftUI view reads data with `@Query` (Chapter 4) and writes through a `ModelContext` that SwiftUI puts in the environment. So this chapter is shorter, and the repository is smaller: it holds only the *write* rules.

JOB	ANDROID (ROOM + HILT)	IOS (SWIFTDATA)
Opening the database	`AppDatabase` built in a Hilt module	A `ModelContainer` built once, attached to the app
Reading	DAO methods returning `Flow`	`@Query` in the view (Chapter 4)
Writing	DAO `@Insert` / `@Update` / `@Delete`	`context.insert`, changing a property, `context.delete`
Getting the connection to a screen	Hilt injection	`@Environment(\.modelContext)`

JOB	ANDROID (ROOM + HILT)	IOS (SWIFTDATA)
Several writes as one unit	`db.withTransaction { }`	Make the changes, then `try context.save()` once
Seeding	`ensureSeeded()` on startup	The same function, on startup
Previews	Preview with made-up data	Preview with an in-memory container

Step 1: Add an Order to ItemType

One correction to Chapter 2. Room sorted the types by their auto-generated id, which follows insertion order. SwiftData gives you no such id to sort by, and a query with no sort has no guaranteed order, so the four types need an explicit `sortOrder`. Change `ItemType` to:

```
@Model
final class ItemType {
    @Attribute(.unique) var name: String
    var sortOrder: Int

    @Relationship(deleteRule: .deny, inverse: \Item.type)
    var items: [Item] = []

    init(name: String, sortOrder: Int) {
        self.name = name
        self.sortOrder = sortOrder
    }
}
```

You are changing a stored property, which on an installed app would need a migration. You have nothing installed that matters yet, so simply **delete the app from the simulator** (long-press its icon and remove it) before the next run. This is the same lesson as Android's "raise the database version": once real data exists, model changes are expensive, which is why Chapter 2 was careful.

Step 2: The Container

A `ModelContainer` is the object that owns the store on disk. It is created from a *schema*, the list of model classes. Put the list in one place, because the app, the previews and (in Chapter 10) the

tests all need the same list. Create `Persistence.swift`:

```
import Foundation
import SwiftData

enum Persistence {
    /// Every model the store knows about. Add new models here, and only here.
    static let schema = Schema([Item.self, ItemType.self, Tag.self])

    /// A container on disk for the real app, or in memory for previews and tests.
    static func makeContainer(inMemory: Bool = false) throws -> ModelContainer {
        let configuration = ModelConfiguration(schema: schema, isStoredInMemoryOnly: inM
        return try ModelContainer(for: schema, configurations: [configuration])
    }
}
```

`isStoredInMemoryOnly: true` is what the tutorials in my search use for previews: the store lives only in memory, so nothing is written to disk and nothing survives. Now the app file. If the wizard generated its own `sharedModelContainer` code, replace it with this:

```
import SwiftUI
import SwiftData

@main
struct CatalogueApp: App {
    private let container: ModelContainer

    init() {
        do {
            container = try Persistence.makeContainer()
        } catch {
            // Placeholder. Chapter 9 replaces this with a screen that explains the prob
            fatalError("Could not open the catalogue store: \(error)")
        }
    }

    var body: some Scene {
        WindowGroup {
            RootView()
        }
        .modelContainer(container)
```

```
    }
}
```

`.modelContainer(container)` attaches the container to the window, which does two things: every view inside can use `@Query`, and each gets a `ModelContext` from the environment. (A shorter `.modelContainer(for: [Item.self, ...])` exists and builds the container for you, but then you cannot reuse the same container setup for previews and tests.) A `fatalError` if the store will not open is crude, but at this point in a launch there is nothing sensible to show; it is a known placeholder, not the final design.

Step 3: The Repository

A `ModelContext` is SwiftData's unit of work: you insert and change objects in it, and it writes them to the store when saved. The repository is a small struct wrapping one. Create `CatalogueRepository.swift`:

```
import Foundation
import SwiftData

@MainActor
struct CatalogueRepository {
    let context: ModelContext

    // MARK: Item types

    /// Call at every launch. Inserts the four types only if there are none.
    func ensureSeeded() throws {
        let existing = try context.fetchCount(FetchDescriptor<ItemType>())
        guard existing == 0 else { return }
        for (index, name) in ["Book", "CD", "DVD", "Blu-ray"].enumerated() {
            context.insert(ItemType(name: name, sortOrder: index))
        }
        try context.save()
    }

    func itemTypes() throws -> [ItemType] {
        try context.fetch(FetchDescriptor<ItemType>(sortBy: [SortDescriptor(\.sortOrder)
    }

    // MARK: Tags
```

```
    /// The existing tag with this name (in any case), or a new one. Never inserts a dup
    func tag(named rawName: String) throws -> Tag? {
        let cleaned = Tag.cleaned(rawName)
        guard !cleaned.isEmpty else { return nil }

        let key = Tag.key(for: cleaned)            // a local constant: predicates canno
        var descriptor = FetchDescriptor<Tag>(predicate: #Predicate { $0.key == key })
        descriptor.fetchLimit = 1
        if let found = try context.fetch(descriptor).first { return found }

        let created = Tag(name: cleaned)
        context.insert(created)
        return created
    }

    // MARK: Items

    /// Saves a new or edited item together with its tags, in one save.
    func save(_ item: Item, tagNames: [String]) throws {
        if item.modelContext == nil {
            context.insert(item)                   // insert BEFORE setting relationshi
        }
        var tags: [Tag] = []
        for name in tagNames {
            if let tag = try tag(named: name), !tags.contains(where: { $0.key == tag.key
                tags.append(tag)
            }
        }
        item.tags = tags
        try context.save()
    }

    func delete(_ item: Item) throws {
        context.delete(item)
        try context.save()
    }
}
```

Four things in that file deserve explanation.

- **Why `tag(named:)` looks before it inserts.** Chapter 2's third exercise warned that, according to a secondary source, inserting a model whose `@Attribute(.unique)` value already exists

may *update* the existing record instead of failing. Looking up by key first means we never rely on that: an existing tag is returned untouched, and its spelling stays as first typed.

- **The local `key` constant.** The `#Predicate` macro builds a query that SwiftData translates for the store, so it can only use simple expressions and values captured from outside. Calling `Tag.key(for:)` *inside* the predicate would not be allowed; computing it first and capturing the constant is the usual approach.
- **"Insert before relationships."** A common recommendation in tutorials and forum posts is to insert a new model into the context *before* assigning relationships to it, which `save(_:tagNames:)` does. I could not confirm it from Apple's documentation, but it costs nothing.
- **No separate transaction.** Everything between the first change and `context.save()` is one unit of work: nothing reaches the store until `save()`. That is the SwiftData counterpart of Chapter 7's `withTransaction` on Android, but it depends on *you* calling `save()` once at the end, which is why the repository does it for every write. (SwiftData's main context may also save automatically; the unit of work still ends when the function returns, because nothing here awaits.)

Step 4: Seeding at Launch

Seeding needs a context, and the cleanest place to get one is a view, via the environment. Create a `RootView` that every later screen will sit inside, and that seeds as it appears:

```
import SwiftUI
import SwiftData

struct RootView: View {
    @Environment(\.modelContext) private var context

    var body: some View {
        ContentView()
            .task {
                do {
                    try CatalogueRepository(context: context).ensureSeeded()
                } catch {
                    print("Seeding failed: \(error)")     // Chapter 9 shows this to the
                }
            }
```

```
    }
}
```

`.task` runs when the view appears. Because screens read with `@Query`, which updates automatically, it does not matter that seeding finishes a moment after the first frame: the type chips and the type picker simply fill in. The repository is a cheap struct, so each screen can make one from its own `modelContext` when it needs one; there is no dependency-injection library and no singleton.

Step 5: Previews on an In-Memory Store

A SwiftUI preview runs your view in Xcode's canvas, and a view that uses `@Query` needs a container to run at all. Never point a preview at the real store. The tutorials in my search agree on a pattern: build *one shared* in-memory container, fill it with sample data, and pass it to each preview. They also report that creating a fresh container inside every `#Preview` block can crash the canvas, which is the reason for sharing one.

```
import Foundation
import SwiftData

@MainActor
enum PreviewData {
    /// One shared, seeded, in-memory container for every preview.
    static let container: ModelContainer = {
        let container = try! Persistence.makeContainer(inMemory: true)
        let repo = CatalogueRepository(context: container.mainContext)
        try! repo.ensureSeeded()

        let types = try! repo.itemTypes()            // Book, CD, DVD, Blu-ray
        let samples: [(String, String, Int, Int, [String])] = [
            ("Dune", "Frank Herbert", 1965, 0, ["sci-fi", "classic"]),
            ("Kind of Blue", "Miles Davis", 1959, 1, ["jazz"]),
            ("Alien", "Ridley Scott", 1979, 2, ["sci-fi"])
        ]
        for (title, creator, year, typeIndex, tags) in samples {
            let item = Item(title: title, creator: creator, releaseYear: year, type: typ
            try! repo.save(item, tagNames: tags)
        }
        return container
```

```
    }()
}
```

The `try!` is acceptable here and nowhere else: this code only ever runs in the canvas, and a crash there is the quickest way to tell you the model is wrong. Use it in any preview like this:

```
#Preview {
    ContentView()
        .modelContainer(PreviewData.container)
}
```

The sample data also runs the repository's own code (seeding, `tag(named:)`, saving with relationships), so the canvas becomes a quick, free test of the Chapter 2 relationship: if the many-to-many inverse is wrong, this is where it should fail, loudly, before you have written a single screen.

Step 6: See It Work

You have no list screen yet, so replace `ContentView` with a temporary one that shows the stored item types. It uses `@Query`, the way Chapter 4's list will:

```
import SwiftUI
import SwiftData

// TEMPORARY: Chapter 4 replaces this with the real list screen.
struct ContentView: View {
    @Query(sort: \ItemType.sortOrder) private var types: [ItemType]
    @Query private var tags: [Tag]

    var body: some View {
        List {
            Section("Item types") {
                ForEach(types) { Text($0.name) }
            }
            Section("Tags") {
                ForEach(tags) { Text($0.name) }
            }
        }
```

```
    }
}

#Preview {
    ContentView()
        .modelContainer(PreviewData.container)
}
```

Run in the simulator: you should see Book, CD, DVD and Blu-ray, in that order, and an empty tags section. Quit and run again: still four types, not eight, because `ensureSeeded()` counts first. The canvas preview should additionally show the three sample tags, *sci-fi*, *classic* and *jazz*, with *sci-fi* appearing once even though two items use it.

COMPARE WITH THE ANDROID COURSE

In *Personal Catalogue: Android (Kotlin & Compose)* Chapter 3, this layer was five Kotlin files plus Gradle and Hilt setup, and the compiler checked the queries. Here it is three small files and a couple of modifiers, but the checking happens at run time, and the sample-data preview is how you catch mistakes early. The repository functions are the same ideas in both courses: seed on launch, find-or-create a tag by its normalised name, and save an item with its tags as one unit.

Challenges

EXERCISE 1

Make all the changes (`sortOrder`, `Persistence`, the app file, the repository, `RootView`, `PreviewData`, the temporary `ContentView`), delete the old app from the simulator, and run. Record what you see on first launch, after a relaunch, and in the canvas preview. List every build or run-time error you had and what caused it.

View solution

EXERCISE 2

Prove that `tag(named:)` never creates duplicates. Add a temporary button to `ContentView` that calls `tag(named:)` with "Sci-Fi", then "SCI-FI", then " sci-fi " and saves. Predict how many tags exist and what name it shows, then run it and compare. Explain what would be different if the function simply did `context.insert(Tag(name: name))`.

View solution

EXERCISE 3

Chapter 2 gave `ItemType.items` the delete rule `.deny`, and its sources never confirmed it. Test it now: in the preview or a temporary button, try to delete the type "Book" while "Dune" uses it. Record exactly what happens (an error, a crash, nothing, or the type deleted anyway) and when it happens (at `delete` or at `save`). Then say what the repository should do about it.

View solution

CHAPTER 3 QUICK REFERENCE

- Container: `ModelContainer(for: schema, configurations: [ModelConfiguration(schema:, isStoredInMemoryOnly:)])`; attach with `.modelContainer(container)`; keep the model list in one place (`Persistence.schema`)
- Context: `@Environment(\.modelContext)`; write with `insert`, property changes, `delete`, then `try context.save()`
- Read once: `context.fetch(FetchDescriptor<T>(predicate:, sortBy:))`, `fetchCount`; read live in views with `@Query` (Chapter 4)
- Predicates use captured constants, not calls to your own functions: `let key = Tag.key(for: name); #Predicate { $0.key == key }`
- Insert a new model before assigning its relationships
- Seed idempotently: count first, insert only when empty; give types an explicit `sortOrder`
- Find-or-create tags by key; never insert a tag without looking first
- Previews: one shared in-memory, seeded container (`PreviewData.container`), never the real store

NOT YET VERIFIED

Checked (one web search on 2026-11-19), through tutorials and forum posts only: that a `ModelConfiguration` with in-memory storage keeps preview data off disk; that the common preview pattern is a shared, `@MainActor` container filled via `mainContext` and passed with `.modelContainer(...)`; that one tutorial checks whether the store is empty with a `FetchDescriptor` before seeding; that creating a new container inside each `#Preview` has been reported to crash; and that on macOS with iCloud an in-memory setting needed `cloudKitDatabase: .none` (not relevant to an iPhone app with no iCloud).

Not checked, from general knowledge: `ModelContext.save()`, `fetchCount`, `insert` and `delete` semantics; `fetchLimit`; `#Predicate` capturing a local constant; the exact initialisers of `Schema`, `ModelConfiguration` and `ModelContainer` in your SDK; that `modelContainer(_:)` accepts a container instance; whether the main context autosaves in a way that could save part of a multi-step change; "insert before assigning relationships" (a tutorial convention, not a documented rule); `@MainActor` and Swift-concurrency behaviour of `App.init` and `mainContext` under your Xcode's language mode; that `@Model` types work as `ForEach` content without an explicit `id`; the `.deny` rule (see Exercise 3); and all code, **none of which was compiled**. I did not consult Apple's reference pages for any of these APIs.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19), none of them Apple reference pages: AppCoda: SwiftData previews; Kodeco: Data Persistence with SwiftData; Apple Developer Forums: SwiftData preview sample code; Apple Developer Forums: Xcode 15 preview crashes; SwiftData ModelContainer setup (third-party article); Solving SwiftData preview crashes (third-party article). Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: iOS (Swift & SwiftUI) 4 builds the list screen: an `@Query` sorted newest first, the type filter, rows showing title, creator and tags, an empty state, and the `@Observable` view model that holds the filter choice.

Personal Catalogue: iOS (Swift & SwiftUI)

Course 1 · Chapter 4 · @Query, @Observable and Filter Chips

The store exists but nothing shows it. This chapter builds the app's home screen: a scrolling list of every item, newest first, with a row of chips to filter by type (Book, CD, DVD, Blu-ray), and a friendly message when there is nothing to show. It also sets the pattern every later screen follows: **SwiftData supplies the data with `@Query`, a small `@Observable` class holds the choices the user makes, and the view draws both.**

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Nothing in this chapter was compiled or run** (no Mac or Xcode here). My search found tutorials, forum posts and one Apple documentation page, "Filtering and sorting persistent data", on `@Query` with predicates and sorting. It found **nothing on `@Observable`, `ContentUnavailableView` or the list/ForEach combination**, so those are from my memory of iOS 17 and later, and the code is the place to expect compile errors.

The Pattern Compared with Android

PART	ANDROID (CHAPTER 4)	IOS (THIS CHAPTER)
Where the data comes from	Repository `Flow`, collected by the view model	`@Query` in the view; SwiftUI refreshes it when the store changes
Where the user's choices live	`MutableStateFlow` in the `ViewModel`	Properties of an `@Observable` class held in `@State`
How a screen is split	Stateful composable and stateless composable	One view, with the logic that can be tested moved into the view model
The scrolling list	`LazyColumn`	`List` with `ForEach`
Filter chips	`FilterChip` in a `LazyRow`	Buttons styled as chips, in a horizontal `ScrollView`
Empty state	A `Box` with a message	`ContentUnavailableView`

The split is much lighter on iOS. The Android view model had to combine three flows and turn them into state; here SwiftUI does the "collect and redraw" part for you, so the view model

shrinks to the user's choices plus any logic worth testing.

Step 1: The View Model

`@Observable` (from the Observation framework, iOS 17 and later) makes a class's properties trackable: any SwiftUI view that reads one is redrawn when it changes, with no `@Published` or `ObservableObject` needed. For now the model holds one choice, the selected type, plus a function that applies it. Create `ListViewModel.swift`:

```
import Foundation
import Observation

@Observable
final class ListViewModel {
    /// The name of the selected item type; nil means "All".
    var selectedTypeName: String?

    /// Applies the type filter to an already-sorted list.
    func filtered(_ items: [Item]) -> [Item] {
        guard let name = selectedTypeName else { return items }
        return items.filter { $0.type?.name == name }
    }
}
```

Be honest about why this exists: for a single optional property, a plain `@State private var selectedTypeName: String?` in the view would do. It is a class here for three reasons: it mirrors the Android course, Chapter 8 adds the search text and the tag filter to it, and `filtered(_:)` is plain Swift that Chapter 10 can test without any UI.

Step 2: The Query

`@Query` is a property wrapper that fetches models and keeps the result current. Sorting is given as a key path and an order:

```
@Query(sort: \Item.createdAt, order: .reverse) private var items: [Item]
@Query(sort: \ItemType.sortOrder) private var types: [ItemType]
```

The first line is "every item, newest first". Add or delete an item anywhere in the app and `items` changes by itself: that is the equivalent of Room's `Flow` from the Android course.

Why filter in memory rather than in the query? A query can take a `#Predicate`, and the sources I found show the usual way to make it depend on a choice: build the `Query` in the view's own `init`, passing the choice in. But that pattern has costs: the `@Query` has to be assigned in a custom initialiser, an Apple forum reply says SwiftData cannot tell whether the predicate changed and so re-fetches whenever that `init` runs, and one of the search results is a thread about a runtime crash with a predicate. Filtering a handful of types out of an array in Swift is simpler and safe, exactly as it was on Android in this chapter. Chapter 8, where search and tag filters need the database, uses the dynamic-query pattern properly.

Step 3: The Screen

Create `ListView.swift`. The view owns the model with `@State`; that matters, and Exercise 3 asks you to explain why.

```
import SwiftUI
import SwiftData

struct ListView: View {
    @Environment(\.modelContext) private var context
    @Query(sort: \Item.createdAt, order: .reverse) private var items: [Item]
    @Query(sort: \ItemType.sortOrder) private var types: [ItemType]
    @State private var model = ListViewModel()

    private var shownItems: [Item] { model.filtered(items) }

    var body: some View {
        NavigationStack {
            VStack(spacing: 0) {
                TypeFilterRow(types: types, selected: $model.selectedTypeName)

                if items.isEmpty {
                    ContentUnavailableView(
                        "Your catalogue is empty",
                        systemImage: "books.vertical",
                        description: Text("Tap + to add your first item.")
                    )
                } else if shownItems.isEmpty {
```

```
                    ContentUnavailableView(
                        "Nothing of this type yet",
                        systemImage: "line.3.horizontal.decrease.circle",
                        description: Text("Choose All to see everything.")
                    )
                } else {
                    List {
                        ForEach(shownItems) { item in
                            ItemRow(item: item)
                        }
                    }
                }
            }
            .navigationTitle("My Catalogue")
            .toolbar {
                ToolbarItem(placement: .primaryAction) {
                    Button("Add", systemImage: "plus") { addSampleItem() }   // TEMPORAR
                }
            }
        }
    }

    // TEMPORARY: lets you see rows before the form exists (Chapter 6 removes this).
    private func addSampleItem() {
        guard let type = types.first else { return }
        let item = Item(title: "Sample item", creator: "Sample creator", type: type)
        try? CatalogueRepository(context: context).save(item, tagNames: [])
    }
}
```

Points worth knowing. `$model.selectedTypeName` is a *binding* to the property: the chips read it and write it, and writing redraws the view. `ForEach(shownItems)` needs no explicit id, because every `@Model` class is identifiable (the assumption flagged in Chapter 3). `List` only builds rows as they scroll into view, so it suits a list of unknown length, as `LazyColumn` did on Android. The two empty messages come from comparing `items` (everything) with `shownItems` (after the filter): "Your catalogue is empty" only when there really is nothing, "Nothing of this type yet" when the filter hides everything. `NavigationStack` is here only to provide the title bar and toolbar; Chapter 5 gives it destinations.

Step 4: The Chips and the Rows

SwiftUI's built-in controls do not include a chip, so build one from a `Button` with a capsule background. `TypeFilterRow` puts an "All" chip and one per type in a horizontal `ScrollView`:

```
import SwiftUI

struct TypeFilterRow: View {
    let types: [ItemType]
    @Binding var selected: String?

    var body: some View {
        ScrollView(.horizontal, showsIndicators: false) {
            HStack(spacing: 8) {
                FilterChip(title: "All", isSelected: selected == nil) { selected = nil }
                ForEach(types) { type in
                    FilterChip(title: type.name, isSelected: selected == type.name) { se
                }
            }
            .padding(.horizontal)
            .padding(.vertical, 8)
        }
    }
}

struct FilterChip: View {
    let title: String
    let isSelected: Bool
    let action: () -> Void

    var body: some View {
        Button(action: action) {
            Text(title)
                .font(.subheadline)
                .padding(.horizontal, 14)
                .padding(.vertical, 7)
                .foregroundStyle(isSelected ? Color.white : Color.primary)
                .background(isSelected ? Color.accentColor : Color(.secondarySystemBackg
        }
        .buttonStyle(.plain)
        .accessibilityAddTraits(isSelected ? .isSelected : [])
    }
}
```

The colours are system and semantic ones (`.primary` , `.accentColor` , `secondarySystemBackground`), never fixed values, so they adapt to light and dark appearance without any extra work: Chapter 9 relies on that. The `isSelected` accessibility trait lets VoiceOver say which chip is chosen. Each row is a small view of its own:

```
import SwiftUI

struct ItemRow: View {
    let item: Item

    private var byline: String {
        [item.creator.isEmpty ? nil : item.creator, item.releaseYear.map(String.init)]
            .compactMap { $0 }
            .joined(separator: " · ")
    }

    var body: some View {
        VStack(alignment: .leading, spacing: 4) {
            Text(item.title)
                .font(.headline)
                .lineLimit(2)
            if !byline.isEmpty {
                Text(byline)
                    .font(.subheadline)
                    .foregroundStyle(.secondary)
            }
            Text(([item.typeName] + item.sortedTags.map(\.name)).joined(separator: " · "
                .font(.caption)
                .foregroundStyle(.tint)
        }
        .padding(.vertical, 2)
    }
}
```

`typeName` and `sortedTags` are the helper properties from Chapter 2, which is exactly what they were for. `item.releaseYear.map(String.init)` turns an optional year into an optional string, so a missing year simply disappears from the byline.

Step 5: Wiring and Previews

Make `RootView` show the new screen instead of the temporary one, and delete the temporary `ContentView` from Chapter 3 (change `ContentView()` in `RootView` to `ListView()`). Add an empty-store container to `PreviewData` so the empty state can be previewed too, and use it:

```
// In PreviewData:
static let emptyContainer: ModelContainer = {
    let container = try! Persistence.makeContainer(inMemory: true)
    try! CatalogueRepository(context: container.mainContext).ensureSeeded()
    return container
}()

// At the bottom of ListView.swift:
#Preview("With sample data") {
    ListView().modelContainer(PreviewData.container)
}

#Preview("Empty catalogue") {
    ListView().modelContainer(PreviewData.emptyContainer)
}
```

Both containers are created once and shared, as Chapter 3 explained. Run the app: you should see "Your catalogue is empty". Tap "+" a few times and sample rows appear immediately, newest first, with no refresh code: `@Query` noticed the store change. Tap a type chip that no sample item uses (only the first type, Book, is used) to see the second empty message.

COMPARE WITH THE ANDROID COURSE

Personal Catalogue: Android (Kotlin & Compose) Chapter 4 needed a view model that combined three `Flow`s into one state object, and a split between a stateful and a stateless composable. On iOS the "collect and redraw" machinery is built in, so the same screen is one view plus a tiny model. The cost is that you have less control: you cannot easily see, or change, exactly when SwiftData re-fetches. Both versions end with the same behaviour: items appear and disappear as the data changes, and the type filter and the two empty states work the same way.

Challenges

EXERCISE 1

Add `ListViewModel`, `ListView`, `TypeFilterRow`, `FilterChip` and `ItemRow`, update `RootView` and the previews, and run it. Tap "+" three times, tap the CD chip, then All. Write down what you see for each of the two empty messages and how you triggered each.

View solution

EXERCISE 2

Change the navigation title to show the number of items currently listed, for example "My Catalogue (3)". Decide where the number should be computed: in the view, or as a property of `ListViewModel`.

View solution

EXERCISE 3

The model is stored with `@State private var model = ListViewModel()`. Explain what would go wrong with `private let model = ListViewModel()` instead, and why the chip selection survives the user rotating the phone. Then say whether the selection survives the app being quit and reopened, and name a property wrapper that would make it do so.

View solution

CHAPTER 4 QUICK REFERENCE

- `@Query(sort: \Item.createdAt, order: .reverse) private var items: [Item]` fetches and keeps itself current
- `@Observable final class` (iOS 17+) holds user choices; own it with `@State private var model = ...`; bind with `$model.property`
- Filter in memory for a simple choice; for database-side filtering build the `Query` in the view's `init` (Chapter 8)
- `List { ForEach(items) { ... } }` builds rows lazily; `@Model` classes are identifiable
- Chip: a `Button` with `.buttonStyle(.plain)` and a `Capsule` background, using semantic colours and the `.isSelected` accessibility trait
- `ContentUnavailableView("Title", systemImage:, description:)` for empty states; two of them: no items at all, and no items of the chosen type

- Previews: `#Preview("name") { ListView().modelContainer(PreviewData.container) }` with shared containers

NOT YET VERIFIED

Checked (one web search on 2026-11-19): that the usual way to make a `@Query` depend on a value is to assign it in the view's `init` (`_items = Query(filter:sort:)`), from a Stack Overflow answer and Hacking with Swift material; that a search tutorial relies on the view being re-initialised when the search text changes; and, from an Apple forum reply, that SwiftData assumes a re-fetch each time that `init` runs because it cannot tell whether the predicate changed. The search also surfaced Apple's page "Filtering and sorting persistent data" and a thread about a runtime crash with the predicate macro, neither of which I read in detail.
Not checked, from general knowledge: `@Query(sort:order:)` with a key path and `.reverse`; `@Observable` and `@State` ownership on iOS 17 and later; `ContentUnavailableView` and its initialiser; `Button(_:systemImage:)`; `ToolbarItem(placement: .primaryAction)`; `.foregroundStyle(.tint)`; `.accessibilityAddTraits(.isSelected)`; that `@Model` classes need no explicit `id` in `ForEach`; and all code, **none of which was compiled**. I did not consult Apple's reference pages for any of these APIs.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Apple Developer: Filtering and sorting persistent data (not read in detail); Create with Swift: Performing search with SwiftData in a SwiftUI app; DEV Community: SwiftData dynamically query filtering; How to dynamically change a query's sort order or predicate (a copy of a Hacking with Swift article); Apple Developer Forums: Using any SwiftData Query causes app to hang. Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: iOS (Swift & SwiftUI) 5 adds the detail screen: navigating from a row with `NavigationStack` and `navigationDestination`, the detail layout, and deleting an item after a confirmation.

Personal Catalogue: iOS (Swift & SwiftUI)

Course 1 · Chapter 5 · Navigation, Detail and a Safe Delete

The list works, but tapping a row does nothing. This chapter adds a second screen that shows one item in full, moves between the two with **NavigationStack**, and lets you delete an item after a confirmation. The navigation part is easy. The delete part has a real trap, which the forum threads in my search describe again and again: **a view that still holds a reference to a deleted SwiftData model can crash the app**. Most of this chapter's care goes there.

> **WHAT WAS RUN, AND WHAT WAS NOT, AND WHAT THE SOURCES SAY**
>
> **Nothing in this chapter was compiled or run** (no Mac or Xcode here). My first search confirmed value-based navigation from Apple's documentation pages and forum threads. It found **nothing on** `confirmationDialog`. My second search found a family of Apple-forum crash reports about deleted models ("backing data was detached", "invalidated", "never access a full future backing data") but **no source that says which delete-and-dismiss order is safe**. So the ordering used here is a reasoned precaution, and Exercise 3 asks you to *test* it on your own Xcode instead of trusting me.

Navigation in One Table

IDEA	ANDROID (NAVIGATION COMPOSE)	IOS (NAVIGATIONSTACK)
The container	`NavHost` with a route table	`NavigationStack`
A screen's address	A route object, `DetailRoute(itemId: 7)`	A *value* pushed on the stack: here the `Item` itself
Going there	`navController.navigate(DetailRoute(id))`	`NavigationLink(value: item) { ... }`
The route table	`composable<DetailRoute> { ... }`	`.navigationDestination(for: Item.self) { item in ... }`
Back	`popBackStack()`	The back button, or `@Environment(\.dismiss)`
What is passed	An id; the screen loads the item	The live `Item` object

That last row is the important difference. On Android, the detail screen was handed an id and loaded the item itself, so it needed Loading and NotFound states, and a deleted item simply became "not found". On iOS you hand over the *object*. That is convenient (an `@Model` object is observable, so the detail screen updates by itself when the item is edited, and can never be stale), but it is exactly what makes deletion dangerous: if the object is deleted while the screen holds it, the next time the screen reads a property there may be nothing behind it.

Step 1: Link and Destination

Apple's documentation describes `NavigationLink(value:)`, which pushes a *value* onto the stack; the value's type must be `Hashable`. A separate modifier, `navigationDestination(for:)`, says which view to build for a value of that type. A SwiftData `@Model` class is normally `Hashable` already, and one result in my search shows a SwiftData class used directly as a link value, but **I could not confirm the conformance from a source**: if Xcode says `Item` does not conform to `Hashable`, push `item.persistentModelID` instead and look the item up in the destination. Two changes to `ListView`:

```
// 1. In the List, wrap each row in a link that carries the item:
List {
    ForEach(shownItems) { item in
        NavigationLink(value: item) {
            ItemRow(item: item)
        }
    }
}

// 2. On the VStack inside the NavigationStack (next to .navigationTitle), say where an
.navigationDestination(for: Item.self) { item in
    DetailView(item: item)
}
```

Where the destination modifier sits matters. One forum answer in my search found a link failing because its matching `navigationDestination` was not visible from where the link was, and Apple's documentation says the modifier belongs on a nearby view. Put it on the container, not on something inside the `List` that is created lazily for each row. Also do **not** give `DetailView` its own `NavigationStack`: another forum thread reports a nested stack breaking navigation after data changes, and the detail screen is already inside the list's stack.

Step 2: The Detail Screen

Create `DetailView.swift`. The layout is a `List` with one section of labelled values; `LabeledContent` is the built-in "label on the left, value on the right" row. A small helper skips blank values, as the Android version's `Field` did.

```
import SwiftUI
import SwiftData

struct DetailView: View {
    let item: Item

    @Environment(\.modelContext) private var context
    @Environment(\.dismiss) private var dismiss
    @State private var showConfirm = false

    var body: some View {
        // Never read a deleted model: draw nothing while the screen is going away.
        if item.isDeleted {
            Color.clear
        } else {
            List {
                Section {
                    Text(item.title)
                        .font(.title2.bold())
                    DetailField("Type", item.typeName)
                    DetailField("Creator", item.creator)
                    DetailField("Format", item.formatDetail)
                    DetailField("Year", item.releaseYear.map(String.init) ?? "")
                    DetailField("Tags", item.sortedTags.map(\.name).joined(separator: ",
                    DetailField("Notes", item.notes)
                    DetailField("Added", item.createdAt.formatted(date: .abbreviated, ti
                }
            }
            .navigationTitle("Details")
            .navigationBarTitleDisplayMode(.inline)
            .toolbar {
                ToolbarItemGroup(placement: .primaryAction) {
                    Button("Edit", systemImage: "pencil") { /* Chapter 6 */ }
                    Button("Delete", systemImage: "trash", role: .destructive) { showCon
                }
            }
            .confirmationDialog("Delete this item?", isPresented: $showConfirm, titleVis
```

```
                Button("Delete", role: .destructive) { deleteItem() }
                Button("Cancel", role: .cancel) { }
            } message: {
                Text("\"\(item.title)\" will be removed from your catalogue. This cannot
            }
        }
    }

    private func deleteItem() {
        let doomed = item
        dismiss()                                // 1. leave the screen first
        Task { @MainActor in                         // 2. delete on the next turn of the r
            do {
                try CatalogueRepository(context: context).delete(doomed)
            } catch {
                print("Delete failed: \(error)")       // Chapter 9 shows this to the us
            }
        }
    }
}

private struct DetailField: View {
    let label: String
    let value: String

    init(_ label: String, _ value: String) {
        self.label = label
        self.value = value
    }

    var body: some View {
        if !value.isEmpty {
            LabeledContent(label, value: value)
        }
    }
}
```

Why the Delete Looks Like That

The crash reports in my search share a shape: some view still holds a deleted model and reads it. One poster suspects a child view re-renders *before* the query updates; another reply says that

clearing stale references before the delete avoids the crash; a third suggests saving first so the backing data exists. None confirms a fix, so `deleteItem` layers three precautions, each cheap:

- **Leave first.** `dismiss()` pops the detail screen, so it stops being the thing that renders the item. The delete then runs in a `Task`, one turn of the main loop later, so the pop has started before the model disappears. This is an ordering *precaution*, not a guarantee: `Task` only schedules, and I could find no source that confirms the timing.
- **Hold the model in a local.** `let doomed = item` keeps the object alive for the delete even after the view has gone.
- **Draw nothing for a deleted model.** The `if item.isDeleted` guard stops the body reading properties of a deleted item if SwiftUI re-renders in between. (I believe `isDeleted` is a property of SwiftData models; Xcode will tell you at once if not, and Exercise 3 asks you to find the safest equivalent.)

The repository's `delete` (Chapter 3) saves after deleting, which is what makes the change durable. Chapter 2's delete rule means the item simply vanishes from each tag's `items`; the tags themselves are untouched. The list refreshes by itself because `@Query` notices the store changed. The `confirmationDialog` is a sheet that slides up with a destructive button and a Cancel button; marking the delete button `role: .destructive` shows it in red, as iOS users expect.

Step 3: Preview

The preview needs an item from the shared sample store. Add a helper to `PreviewData` and use it inside a stack so the toolbar shows:

```
// In PreviewData:
static var sampleItem: Item {
    try! container.mainContext.fetch(FetchDescriptor<Item>(sortBy: [SortDescriptor(\.tit
}

// At the bottom of DetailView.swift:
#Preview {
    NavigationStack {
        DetailView(item: PreviewData.sampleItem)
    }
```

```
        .modelContainer(PreviewData.container)
    }
```

(The `NavigationStack` here is only for the preview, so the toolbar has a bar to sit in; the real app's stack is in `ListView`.) Now run the app: add a sample item with "+", tap its row, go back with the arrow, open it again and tap the bin, then Cancel, then the bin and Delete. The row should vanish from the list. Watch the console on that last step; if the app crashes, read the message and see Exercise 3.

COMPARE WITH THE ANDROID COURSE

In *Personal Catalogue: Android (Kotlin & Compose)* Chapter 5, the detail screen received an id and exposed Loading, NotFound and Found states, so a deleted item just became "not found" and the screen popped off the stack after the delete. The iOS screen gets the object itself, which removes that whole layer of state but moves the risk to the delete. The two designs are the classic trade-off: passing an id means always reading fresh data and handling absence; passing the object is simpler until the object goes away.

Challenges

EXERCISE 1

Add the link, the destination, `DetailView` and the preview. Run the app and perform the full sequence: add, open, back, delete-with-cancel, delete. Write down whether you needed to change anything (for example the `Hashable` fallback or the `isDeleted` guard), and what Xcode said.

View solution

EXERCISE 2

Add swipe-to-delete on the list rows, with the same confirmation, using `.swipeActions`. Decide where the "which item is waiting to be deleted" state must live, and why a `@State private var showConfirm: Bool` alone is not enough here.

View solution

EXERCISE 3

Test the delete ordering instead of trusting it. Try three versions in turn and record, for each, whether the app crashed, what the console said, and the iOS version: (a) delete, then `dismiss()`, with no guard; (b) `dismiss()` then delete in a `Task`, with no guard; (c) the chapter's version. Then say which you would ship and why.

View solution

CHAPTER 5 QUICK REFERENCE

- Push: `NavigationLink(value: item) { ... }`; the value type must be `Hashable` (fallback: push `item.persistentModelID`)
- Route: `.navigationDestination(for: Item.self) { item in DetailView(item: item) }` on the container, not inside a lazy row; no second `NavigationStack` in the detail view
- Back: system back button, or `@Environment(\.dismiss)`
- `LabeledContent("Label", value: text)` for label-and-value rows; skip blank values
- Confirmation: `.confirmationDialog(title, isPresented: $flag, titleVisibility: .visible) { Button("Delete", role: .destructive) { ... }; Button("Cancel", role: .cancel) { } } message: { ... }`
- Safe delete: keep the model in a local, leave the screen first, delete afterwards, and guard the body against a deleted model
- Deleting an item clears its links; the tags remain (Chapter 2's default rule)

NOT YET VERIFIED

Checked (two web searches on 2026-11-19): from Apple's documentation and forum threads, that `NavigationLink(value:)` needs a `Hashable` value and `navigationDestination(for:)` maps a value type to a view; that a destination must be visible from the link (a forum answer) and belongs on a nearby view (Apple); that a SwiftData class has been used as a link value in a forum question; that a nested `NavigationStack` in a detail view was reported to break navigation after data changes; and that several forum threads report crashes ("backing data was detached", "invalidated", "never access a full future backing data") when a view holds a deleted model, with suggestions to clear stale references or save first.

Not checked, from general knowledge: that `@Model` classes conform to `Hashable`; the `isDeleted` property; whether dismiss-then-delete in a `Task` is actually safe (**no source confirms any order**); `confirmationDialog` and its message closure (the search found nothing on it); `LabeledContent`;

`ToolbarItemGroup`; `Button(_:systemImage:role:)`; `Date.formatted(date:time:)`; `@Environment(\.dismiss)` behaviour inside a pushed view; and all code, **none of which was compiled**. I did not consult Apple's pages for SwiftData deletion.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Apple Developer: Understanding the navigation stack; Apple Developer: NavigationLink init(_:value:); Create with Swift: Programmatic navigation with navigation destination; Apple Developer Forums: NavigationLinks to "Details" page do not work after data changes; Apple Developer Forums: SwiftData error when trying to delete items from a list; Apple Developer Forums: Model instance was invalidated because its backing data could no longer be found; Apple Developer Forums: a "backing data was detached" report. Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: iOS (Swift & SwiftUI) 6 builds the add and edit form: one `Form` whose labels change with the item type, validation, and bindings to a draft copy of the item. It replaces Chapter 4's temporary "+" behaviour and makes the Edit button work.

Personal Catalogue: iOS (Swift & SwiftUI)

Course 1 · Chapter 6 · Form, Draft and Validation

So far the only way to put data in is a temporary "+" that inserts "Sample item". This chapter replaces it with the real thing: **one form screen for both adding and editing**, whose labels change with the item type (a book has an "Author", a CD an "Artist", a film a "Director"), with validation, and with a design that makes **Cancel really discard**. It also finishes two loose ends from Chapter 5: the Edit button will work, and the sample-item code is deleted.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Nothing in this chapter was compiled or run** (no Mac or Xcode here). My search confirmed from Apple's SwiftData documentation that one editor view can serve both adding and editing, with the title switching on whether a model is passed in, and that the context is saved explicitly. It also surfaced forum threads on discarding edits and a "draft" pattern. It found **nothing on the number keyboard or the segmented** `Picker`, which are from my memory, as is every API detail below.

One Form, Two Jobs

Adding and editing are the same screen with different starting values, which a web developer will recognise from Django's `ModelForm`. Apple's own SwiftData sample does the same: a single editor view whose title is "Add" or "Edit" depending on whether it was given an item.

CONCERN	DJANGO MODELFORM	ANDROID (CHAPTER 6)	IOS (THIS CHAPTER)
Create vs update	Two URLs, one form class	One route, `itemId` 0 = new	One view, `item: Item?`, `nil` = new
Where the typed text lives	The request and form object	A `FormUiState` in a view model	An `ItemDraft` struct in `@State`
Field to state	Form field	`value` + `onValueChange`	A binding: `$draft.title`
Validation	`is_valid()`	Derived flags on the state	Derived properties on the draft

CONCERN	DJANGO MODELFORM	ANDROID (CHAPTER 6)	IOS (THIS CHAPTER)
Cancel	Do not save	Leave without saving	Dismiss: nothing was written, because the draft is a copy
Presented as	A page	A navigation destination	A sheet with its own navigation bar

Why a Draft Instead of Editing the Item Directly

SwiftUI makes it tempting to bind a text field straight to the model: with `@Bindable var item: Item`, `TextField("Title", text: $item.title)` edits the stored object on every keystroke. That is the least code, and it has a real flaw for a form with a Cancel button: **the edit has already happened**. My search found Apple-forum threads asking exactly this ("SwiftData and discarding unsaved changes", and one that "sandboxes" edits in a separate model context), and Apple's documentation notes that autosave is a setting, not a certainty, so whether an abandoned edit survives depends on when the context saves. Rather than depend on that, this chapter edits a **copy**: a plain struct, `ItemDraft`, which is only written into the model when the user taps Save. Cancel is then trivially correct, because there is nothing to undo. It is the same idea as the Android course's `FormUiState`.

Step 1: Labels That Follow the Type

Put this in `FieldLabels.swift`. It matches on the seeded type names from Chapter 3, which is fine for a fixed list.

```
struct FieldLabels {
    let creator: String
    let format: String
}

func labels(for typeName: String?) -> FieldLabels {
    switch typeName {
    case "Book":
        return FieldLabels(creator: "Author", format: "Format (hardcover, paperback...)"
    case "CD":
        return FieldLabels(creator: "Artist", format: "Format (album, single...)")
    case "DVD", "Blu-ray":
```

```
            return FieldLabels(creator: "Director", format: "Edition or region code")
        default:
            return FieldLabels(creator: "Creator", format: "Format")
        }
    }
```

Step 2: The Draft

The draft holds every field as text or a simple value, and computes its own validity. Two choices carry over from Android. The year is held as **text** while it is being typed (an empty box is not a number) and converted on save. And the existing tag names travel with the draft: Chapter 7 adds the controls to change them, but until then an edit must *preserve* them, because the repository's `save` replaces an item's tags with the names it is given.

```
import Foundation

struct ItemDraft {
    var title = ""
    var creator = ""
    var formatDetail = ""
    var yearText = ""
    var notes = ""
    var typeName: String?
    var tagNames: [String] = []

    /// A blank draft, for a new item.
    init() {}

    /// A copy of an existing item's fields, for editing.
    init(item: Item) {
        title = item.title
        creator = item.creator
        formatDetail = item.formatDetail
        yearText = item.releaseYear.map(String.init) ?? ""
        notes = item.notes
        typeName = item.type?.name
        tagNames = item.sortedTags.map(\.name)
    }

    var titleError: Bool {
        title.trimmingCharacters(in: .whitespacesAndNewlines).isEmpty
```

```
    }

    var yearError: Bool {
        guard !yearText.isEmpty else { return false }
        let latest = Calendar.current.component(.year, from: .now) + 1
        guard let year = Int(yearText) else { return true }
        return !(1000...latest).contains(year)
    }

    var isValid: Bool { !titleError && !yearError && typeName != nil }
}
```

Because `ItemDraft` is plain Swift with no UI and no database in it, Chapter 10 can test every one of those rules in milliseconds. That is the practical benefit of keeping the logic out of the view.

Step 3: The Form

`ItemFormView` takes an optional item. Its draft is created in `init` from that item (or blank), stored in `@State` so it survives redraws, and each field binds to a piece of it with `$draft.something`. The whole thing is wrapped in its own `NavigationStack`: a *sheet* is a separate presentation, so it needs its own navigation bar for the Cancel and Save buttons. (This is not the nested-stack problem from Chapter 5; that was a stack inside a screen that was already inside one.)

```
import SwiftUI
import SwiftData

struct ItemFormView: View {
    let item: Item?                         // nil = adding a new item

    @Environment(\.modelContext) private var context
    @Environment(\.dismiss) private var dismiss
    @Query(sort: \ItemType.sortOrder) private var types: [ItemType]
    @State private var draft: ItemDraft
    @State private var showErrors = false

    init(item: Item?) {
        self.item = item
        _draft = State(initialValue: item.map(ItemDraft.init(item:)) ?? ItemDraft())
```

```
    }

    var body: some View {
        let labels = labels(for: draft.typeName)

        NavigationStack {
            Form {
                Section("Type") {
                    Picker("Type", selection: $draft.typeName) {
                        ForEach(types) { type in
                            Text(type.name).tag(Optional(type.name))
                        }
                    }
                    .pickerStyle(.segmented)
                }

                Section {
                    TextField("Title (required)", text: $draft.title)
                        .textInputAutocapitalization(.words)
                } footer: {
                    if showErrors && draft.titleError {
                        Text("Please enter a title").foregroundStyle(.red)
                    }
                }

                Section {
                    TextField(labels.creator, text: $draft.creator)
                        .textInputAutocapitalization(.words)
                    TextField(labels.format, text: $draft.formatDetail)
                }

                Section {
                    TextField("Year (optional)", text: $draft.yearText)
                        .keyboardType(.numberPad)
                } footer: {
                    if showErrors && draft.yearError {
                        Text("Enter a four-digit year").foregroundStyle(.red)
                    }
                }

                Section("Notes") {
                    TextField("Notes", text: $draft.notes, axis: .vertical)
                        .lineLimit(3...6)
                }
            }
```

```
            .navigationTitle(item == nil ? "Add Item" : "Edit Item")
            .navigationBarTitleDisplayMode(.inline)
            .toolbar {
                ToolbarItem(placement: .cancellationAction) {
                    Button("Cancel") { dismiss() }
                }
                ToolbarItem(placement: .confirmationAction) {
                    Button("Save") { save() }
                }
            }
            .onChange(of: draft.yearText) { _, newValue in
                let digits = String(newValue.filter(\.isNumber).prefix(4))
                if digits != newValue { draft.yearText = digits }
            }
            .task(id: types.count) {
                if draft.typeName == nil { draft.typeName = types.first?.name }
            }
        }
    }

    private func save() {
        guard draft.isValid else {
            showErrors = true
            return
        }
        guard let type = types.first(where: { $0.name == draft.typeName }) else { return

        let target = item ?? Item(title: draft.title, type: type)
        target.title = draft.title.trimmingCharacters(in: .whitespacesAndNewlines)
        target.creator = draft.creator.trimmingCharacters(in: .whitespacesAndNewlines)
        target.formatDetail = draft.formatDetail.trimmingCharacters(in: .whitespacesAndN
        target.releaseYear = Int(draft.yearText)
        target.notes = draft.notes.trimmingCharacters(in: .whitespacesAndNewlines)
        target.type = type

        do {
            try CatalogueRepository(context: context).save(target, tagNames: draft.tagNa
            dismiss()
        } catch {
            print("Save failed: \(error)")         // Chapter 9 shows this to the user
        }
    }
}

#Preview("Add") {
```

```
    ItemFormView(item: nil).modelContainer(PreviewData.container)
}

#Preview("Edit") {
    ItemFormView(item: PreviewData.sampleItem).modelContainer(PreviewData.container)
}
```

Reading the Form

- **Labels change with the type.** `labels` is recomputed every time `draft.typeName` changes, so tapping a different segment relabels the creator and format fields at once while the text typed stays: the same behaviour as the Android form.
- **The picker's tag.** `draft.typeName` is a `String?`, and a `Picker`'s selection must match its rows' `tag` type exactly, so each row is tagged `Optional(type.name)`. This mismatch (a non-optional tag for an optional selection) is a classic reason a picker looks right and selects nothing.
- **Errors appear after the first Save**, through `showErrors`, so the form does not shout at someone who has not started typing. They show in the section *footer*, the standard place for a hint under a group of fields.
- **The year filter.** `onChange(of:)` strips anything that is not a digit and caps the length at four. (The two-parameter closure is the iOS 17 form.) `.numberPad` shows a digits-only keyboard, but it does not stop a pasted letter, which is why the filter exists.
- **Editing does not touch what the form does not show.** An edit changes the existing object's fields and leaves `createdAt` alone, so the "Added" date is preserved without any extra code: here the live object does for free what the Android version had to remember to carry. The tags are preserved because the draft carries their names back into `save`.
- **New items are not inserted until Save.** `Item(title:type:)` is only created inside `save`, and `CatalogueRepository.save` inserts it (Chapter 3), so pressing Cancel on an Add form leaves no trace.

Step 4: Wiring and Clean-up

Present the form as a sheet from both places that need it, and delete the temporary code.

```swift
// ListView.swift
@State private var showAdd = false                          // add this
// delete: @Environment(\.modelContext) private var context   (no longer used)
// delete: the whole addSampleItem() function

Button("Add", systemImage: "plus") { showAdd = true }       // was { addSampleItem() }

// next to .navigationTitle(...) on the VStack:
.sheet(isPresented: $showAdd) {
    ItemFormView(item: nil)
}

// DetailView.swift
@State private var showEdit = false

Button("Edit", systemImage: "pencil") { showEdit = true }    // was { /* Chapter 6 */ }

// next to .confirmationDialog(...):
.sheet(isPresented: $showEdit) {
    ItemFormView(item: item)
}
```

Run it: tap "+", pick CD (the label becomes "Artist"), type a title, and Save. Open the item, tap the pencil, change the title and Save: the detail screen updates itself with no refresh code, because the item is an observable object and the view reads it. Try Save with an empty title to see the error, and Cancel after typing to confirm nothing is saved.

COMPARE WITH THE ANDROID COURSE

In *Personal Catalogue: Android (Kotlin & Compose)* Chapter 6 the form's state lived in a `ViewModel` and had to survive rotation, and the edit had to carry `createdAt` and the tag ids through by hand. The iOS draft is simpler (a struct in `@State`) and editing a live object keeps untouched fields automatically. The price is the one in this chapter's opening: because the model is live, you must *choose* to edit a copy, or Cancel will not mean cancel. The validation rules and the per-type labels are identical in the two courses.

Challenges

EXERCISE 1

Add `FieldLabels`, `ItemDraft` and `ItemFormView`, make the wiring edits, and remove the temporary code. Add one item of each type and write down the label of the creator field for each. Then edit one and confirm its "Added" date did not change.

View solution

EXERCISE 2

Make the year error appear *immediately* (not just after Save), and make the title field ignore leading spaces as the user types. Say which type each change belongs in (`ItemDraft`, `ItemFormView` or the repository) and why.

View solution

EXERCISE 3

Test the alternative the chapter argued against. In a copy of the form, bind a field straight to the model with `@Bindable var item: Item` and `TextField("Title", text: $item.title)`, with a Cancel button that only dismisses. Edit an item's title, press Cancel, then check the list and quit and relaunch the app. Record what happened each time and the iOS version.

View solution

CHAPTER 6 QUICK REFERENCE

- One form for add and edit: `ItemFormView(item: Item?)`; `nil` means new; title "Add Item" or "Edit Item"
- Edit a **draft copy** (`@State private var draft: ItemDraft`), written to the model only on Save, so Cancel discards for free
- Create the state in `init`: `_draft = State(initialValue: ...)`
- Bindings: `TextField("Title", text: $draft.title)`; year as text, filtered with `onChange(of:)`; `.keyboardType(.numberPad)`
- Segmented `Picker` over an optional: tag each row with `Optional(value)`
- Validation as derived properties on the draft; show errors after the first Save, in section footers

- A sheet needs its own `NavigationStack`; Cancel is `.cancellationAction`, Save is `.confirmationAction`
- Edits must preserve what the form does not show: tags travel in the draft; `createdAt` is kept by editing the live object

NOT YET VERIFIED

Checked (one web search on 2026-11-19): that Apple's SwiftData documentation shows one editor view for adding and editing, titled by whether a model was passed in, and calls `save()` explicitly because autosave is a setting; that Apple-forum threads discuss discarding unsaved edits and sandboxing edits in a separate model context; that a blog describes a "draft" pattern in which discarding is just dismissal; and that Hacking with Swift's book dismisses a SwiftData add form after saving.

Not checked, from general knowledge, and the search found nothing on the keyboard or Picker: `.keyboardType(.numberPad)`; the segmented `Picker` and the `Optional(...)` tag rule; `TextField(axis: .vertical)` and `.lineLimit(3...6)`; `.textInputAutocapitalization`; `onChange(of:)` with two parameters; `Section` with a `footer` closure; `.task(id:)`; the toolbar placements `.cancellationAction` and `.confirmationAction`; that creating an `Item` with a `type` *before* it is inserted into the context is acceptable (Chapter 3 recommended inserting before assigning relationships; if Xcode or the console complains, create the item with `type: nil` and assign it after `save` inserts); the claim in the chapter's opening about when an abandoned live edit would survive; and all code, **none of which was compiled**. I did not consult Apple's reference pages for these APIs.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Apple Developer: Adding and editing persistent data in your app; Apple Developer Forums: SwiftData and discarding unsaved changes; Apple Developer Forums: How to sandbox SwiftData edits in .sheet; Apple Developer Forums: How to do Add, View, Edit (with Cancel and Done); Drafting and discarding edits to Core Data objects in SwiftUI (third-party blog); Hacking with Swift: Creating books with SwiftData. Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: iOS (Swift & SwiftUI) 7 adds tags: a managed list that never allows "Web Development" and "web development" to coexist, suggestions while typing, tag chips, and the many-to-many writes that replace the preserved tag names used in this chapter.

Personal Catalogue: iOS (Swift & SwiftUI)

Course 1 · Chapter 7 · Tag Editor, Suggestions and Chips

Items can have tags in the database, but the form can only preserve the ones they already have. This chapter adds a **tag editor** to the form: type a tag and press return to add it, see suggestions drawn from tags you already use, and remove a tag with one tap. The rules from the shared spec are the point: **"Web Development" and "web development" are the same tag**, and a tag can never appear twice on one item.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Nothing in this chapter was compiled or run** (no Mac or Xcode here). My search was about wrapping chip layouts only: it found several tutorials building one with SwiftUI's `Layout` protocol, plus ready-made packages, so the approach is real. It found **nothing on SwiftData tag relationships or on suggestion-as-you-type fields**; those parts, and every API detail, are from my memory and are the likeliest place for compile errors.

Where the Work Happens

JOB	ANDROID (CHAPTER 7)	IOS (THIS CHAPTER)
Case-insensitive identity	Room `NOCASE` unique index (ASCII only)	A folded `key` property, unique (Chapter 2)
Find or create	Repository transaction	`CatalogueRepository.tag(named:)` (Chapter 3)
Join rows	Explicit cross-reference rows	Assigning `item.tags`; SwiftData writes the join
Wrapping chips	`FlowRow`	A small custom `Layout` (no built-in wrapping stack)
Suggestions	Filtered list under the field	Filtered list under the field

Almost everything database-side already exists. The draft from Chapter 6 carries `tagNames`, and `CatalogueRepository.save(_:tagNames:)` already finds or creates a `Tag` for each name, by its folded key, and assigns them to the item. So the many-to-many write needs *no new code*; this chapter is about the screen that edits the names.

Step 1: A Wrapping Layout

A `HStack` never wraps, and a row of tags needs to. The tutorials in my search build a flow layout by conforming to the `Layout` protocol and implementing two methods: one to report the size, one to place the children. This version is deliberately small; if you would rather not maintain it, the packages in the sources do the same job.

```
import SwiftUI

struct FlowLayout: Layout {
    var spacing: CGFloat = 8

    func sizeThatFits(proposal: ProposedViewSize, subviews: Subviews, cache: inout ()) -
        let maxWidth = proposal.width ?? .infinity
        var x: CGFloat = 0, y: CGFloat = 0, rowHeight: CGFloat = 0, widest: CGFloat = 0
        for view in subviews {
            let size = view.sizeThatFits(.unspecified)
            if x > 0 && x + size.width > maxWidth {
                x = 0; y += rowHeight + spacing; rowHeight = 0
            }
            x += size.width + spacing
            rowHeight = max(rowHeight, size.height)
            widest = max(widest, x - spacing)
        }
        return CGSize(width: widest, height: y + rowHeight)
    }

    func placeSubviews(in bounds: CGRect, proposal: ProposedViewSize, subviews: Subviews
        var x = bounds.minX, y = bounds.minY, rowHeight: CGFloat = 0
        for view in subviews {
            let size = view.sizeThatFits(.unspecified)
            if x > bounds.minX && x + size.width > bounds.maxX {
                x = bounds.minX; y += rowHeight + spacing; rowHeight = 0
            }
            view.place(at: CGPoint(x: x, y: y), proposal: .unspecified)
            x += size.width + spacing
            rowHeight = max(rowHeight, size.height)
        }
    }
}
```

The idea: lay children left to right; when the next one would pass the right edge, start a new row. `sizeThatFits` does a dry run to report the total height, and `placeSubviews` does it again for real.

Step 2: The Tag Editor

`TagEditor` edits a `Binding<[String]>` (the draft's `tagNames`), so it knows nothing about SwiftData writes. It *reads* the existing tags with `@Query` only to offer suggestions. Adding goes through the same folding as storage, so the duplicate rule is enforced in one way everywhere.

```
import SwiftUI
import SwiftData

struct TagEditor: View {
    @Binding var tagNames: [String]
    @Query(sort: \Tag.name) private var allTags: [Tag]
    @State private var input = ""

    // Existing tags matching what is typed, not already on this item.
    private var suggestions: [String] {
        let typed = Tag.key(for: input)
        guard !typed.isEmpty else { return [] }
        let chosen = Set(tagNames.map(Tag.key(for:)))
        return allTags
            .filter { $0.key.contains(typed) && !chosen.contains($0.key) }
            .prefix(5)
            .map(\.name)
    }

    var body: some View {
        VStack(alignment: .leading, spacing: 10) {
            if !tagNames.isEmpty {
                FlowLayout {
                    ForEach(tagNames, id: \.self) { name in
                        TagChip(name: name) { remove(name) }
                    }
                }
            }

            TextField("Add a tag", text: $input)
                .textInputAutocapitalization(.never)
                .submitLabel(.done)
```

```
                .onSubmit { add(input) }

            if !suggestions.isEmpty {
                FlowLayout {
                    ForEach(suggestions, id: \.self) { name in
                        Button(name) { add(name) }
                            .buttonStyle(.bordered)
                            .controlSize(.small)
                    }
                }
            }
        }
    }

    private func add(_ raw: String) {
        let name = Tag.cleaned(raw)
        defer { input = "" }
        guard !name.isEmpty else { return }
        let key = Tag.key(for: name)
        if tagNames.contains(where: { Tag.key(for: $0) == key }) { return }   // already
        // Reuse the stored spelling if the tag exists, so "swift" adds the existing "Sw
        let stored = allTags.first(where: { $0.key == key })?.name
        tagNames.append(stored ?? name)
    }

    private func remove(_ name: String) {
        tagNames.removeAll { $0 == name }
    }
}

struct TagChip: View {
    let name: String
    let onRemove: () -> Void

    var body: some View {
        HStack(spacing: 4) {
            Text(name)
            Button(action: onRemove) {
                Image(systemName: "xmark.circle.fill")
            }
            .buttonStyle(.plain)
            .accessibilityLabel("Remove tag \(name)")
        }
        .font(.subheadline)
        .padding(.horizontal, 10)
```

```
            .padding(.vertical, 5)
            .background(.tint.opacity(0.15), in: Capsule())
        }
    }
```

I assumed `Tag.cleaned(_:)` (trim and collapse spaces) and `Tag.key(for:)` (fold case and diacritics) keep the static signatures from Chapter 2; if yours differ, adjust the calls, not the logic. Three details matter. **Duplicates are decided by the key**, so adding "SWIFT" to an item that already has "Swift" does nothing. **The stored spelling wins**: typing "swift" when "Swift" exists adds "Swift", so the catalogue does not drift into several spellings. And **suggestions never offer what is already chosen**.

Step 3: Put It in the Form

In `ItemFormView`, add a section between the year and the notes. The editor changes `draft.tagNames`, the same array `save()` already hands to the repository, so there is nothing else to change.

```
Section("Tags") {
    TagEditor(tagNames: $draft.tagNames)
}
```

One **problem to watch for**: if the user types a tag and taps Save without pressing return, the text in the field is lost. That is a real usability trap. Exercise 2 asks you to fix it. Also, `ItemRow` and `DetailView` already show `sortedTags`, so tags appear there with no further work.

Tags nobody uses. Remove the last item carrying a tag and the `Tag` stays in the store (the delete rules from Chapter 2 clear only the link). That is harmless for now, and it even keeps the tag as a suggestion. Chapter 8's filter row should show only tags in use; clean-up of orphans is left as a decision for you, in Exercise 3.

> **COMPARE WITH THE ANDROID COURSE**
>
> *Personal Catalogue: Android (Kotlin & Compose)* Chapter 7 needed an explicit cross-reference table and a transaction to keep the join rows consistent, and an ASCII-only `NOCASE` index forced a separate normalised column anyway. On iOS the folded `key` from Chapter 2 does the same job, and assigning `item.tags` is the whole many-to-many write. The cost: the unique-key behaviour of SwiftData on a

duplicate insert is one of the unverified points in this course, which is why the repository finds before it inserts. The wrapping chips are the reverse: Compose gives you `FlowRow`, SwiftUI makes you write the layout.

Challenges

EXERCISE 1

Add `FlowLayout`, `TagEditor` and `TagChip` and the form section. Add the tags "Sci-Fi", "sci-fi" and "SCI-FI" to one item in turn, then open another item and type "sci". Write down what each did and what the suggestion list showed.

View solution

EXERCISE 2

Fix the lost-text trap: if the tag field holds text when Save is pressed, that text should be added as a tag. Decide whether the fix belongs in `TagEditor`, in `ItemFormView` or in `ItemDraft`, and explain what makes the other places awkward.

View solution

EXERCISE 3

Decide what should happen to a tag no item uses any more. Give the arguments for keeping it and for deleting it, choose one, and write the code that implements your choice (where would it run, and what must it check?).

View solution

CHAPTER 7 QUICK REFERENCE

- The editor edits `Binding<[String]>`; the repository's find-or-create (Chapter 3) does the many-to-many write, so the screen needs no database code
- Duplicates are decided by the folded key; reuse the stored spelling of an existing tag

- No built-in wrapping stack: conform to `Layout` (`sizeThatFits` + `placeSubviews`) or use a package
- Return key: `.submitLabel(.done)` + `.onSubmit { }`
- Suggestions: filter `@Query` tags by the typed key, exclude chosen ones, cap the list
- Unused tags stay in the store unless you remove them deliberately

NOT YET VERIFIED

Checked (one web search on 2026-11-19): that SwiftUI has no wrapping stack and that tag-chip layouts are commonly built by conforming to the `Layout` protocol, with ready-made packages (Flow and swiftui-flow-layout) available; one tutorial also touches on tag uniqueness checks, which I did not read.

Not checked, from general knowledge: the exact `Layout` method signatures and the `Void` cache type used above (the likeliest compile error here); `ProposedViewSize` handling of `.infinity`; `.onSubmit` on a `TextField` inside a `Form` section; `.submitLabel`; `.tint.opacity(_:)` as a shape style; `.buttonStyle(.bordered)` with `.controlSize(.small)`; that assigning `item.tags` writes the join without further code; the SwiftData unique-key behaviour (still unverified, as in Chapter 3); and all code, **none of which was compiled**. I did not consult Apple's reference pages.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Swift with Majid: Building custom layout in SwiftUI, basics; DEV Community: Building a wrapping HStack with the SwiftUI Layout protocol; GitHub: toastersocks/flow; GitHub: banghuazhao/swiftui-flow-layout; LogRocket: Implementing tags using SwiftUI (via Plushcap). Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: iOS (Swift & SwiftUI) 8 adds search and filtering: a search field, a tag filter, and the "do I own this?" check, where the dynamic `@Query` pattern from Chapter 4's discussion is finally used.

Personal Catalogue: iOS (Swift & SwiftUI)

Course 1 · Chapter 8 · Search, Tag Filter and "Do I Own This?"

A catalogue is only useful if you can find things in it, and its most common real-world question is "do I already own this?". This chapter adds a **search bar**, a **tag filter** and a **duplicate warning** in the add form. It is also where Chapter 4's promise is kept: the search text goes *into the database query*, using the dynamic-`@Query` pattern that Chapter 4 explained but deliberately avoided.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Nothing in this chapter was compiled or run** (no Mac or Xcode here). My search confirmed that `localizedStandardContains` is the usual case-insensitive match inside a `#Predicate`, that `uppercased()` inside one crashes at runtime, and that **optional to-many relationships in a predicate have caused crashes**. It did **not** cover `.searchable`, and it did not say whether `localizedStandardContains` ignores accents, so that is from memory and flagged below. This chapter's predicate touches only `title` and `creator`, which are plain non-optional strings: that is a deliberate choice, explained next.

What Goes in the Query and What Does Not

FILTER	WHERE IT RUNS	WHY
Search text (title, creator)	In the `@Query` predicate	Plain string properties; the database does the matching and sorting
Type	In memory (Chapter 4)	A handful of items to filter, one optional relationship
Tag	In memory	Searching a to-many relationship inside a predicate is where my sources report crashes; filtering the fetched array is safe

This split is a judgement, not a rule. For a personal catalogue of hundreds of items, filtering the result of a text query in memory costs nothing, and it keeps the risky predicate shapes out of the app. If you ever need the tag filter in the database too, test it carefully on your OS versions first; the forum reports I found suggest making the relationship non-optional (ours already is: `tags: [Tag] = []`).

Step 1: Extend the View Model

```
@Observable
final class ListViewModel {
    var selectedTypeName: String?
    var selectedTagKey: String?          // the folded key of the chosen tag; nil = any
    var searchText = ""

    var hasFilters: Bool {
        selectedTypeName != nil || selectedTagKey != nil || !searchText.isEmpty
    }

    /// Applies the type and tag filters to an already-searched, already-sorted list.
    func filtered(_ items: [Item]) -> [Item] {
        items.filter { item in
            (selectedTypeName == nil || item.type?.name == selectedTypeName) &&
            (selectedTagKey == nil || item.tags.contains { $0.key == selectedTagKey })
        }
    }
}
```

Step 2: A List That Rebuilds Its Query

A `@Query` cannot read a changing value directly, so move the list into its own view whose `init` builds the query from the search text it is given. SwiftUI re-creates that view each time the text changes, which is how the sources describe it working. (An Apple forum reply says SwiftData assumes a re-fetch each time the init runs, so pass in *only* what the query needs, as here.)

```
import SwiftUI
import SwiftData

struct ItemListContent: View {
    @Query private var items: [Item]
    let model: ListViewModel
    let catalogueIsEmpty: Bool
    let searchText: String
    let onAddNew: (String) -> Void

    init(model: ListViewModel, searchText: String, catalogueIsEmpty: Bool, onAddNew: @es
```

```
        self.model = model
        self.searchText = searchText
        self.catalogueIsEmpty = catalogueIsEmpty
        self.onAddNew = onAddNew

        let term = searchText
        let predicate: Predicate<Item> = term.isEmpty
            ? #Predicate<Item> { _ in true }
            : #Predicate<Item> { $0.title.localizedStandardContains(term) || $0.creator.
        _items = Query(filter: predicate, sort: \Item.createdAt, order: .reverse)
    }

    private var shown: [Item] { model.filtered(items) }

    var body: some View {
        if catalogueIsEmpty {
            ContentUnavailableView("Your catalogue is empty", systemImage: "books.vertic
                                   description: Text("Tap + to add your first item."))
        } else if shown.isEmpty {
            VStack(spacing: 12) {
                if !searchText.isEmpty {
                    ContentUnavailableView.search(text: searchText)
                    Button("Add \"\(searchText)\"") { onAddNew(searchText) }
                        .buttonStyle(.borderedProminent)
                } else {
                    ContentUnavailableView("Nothing matches these filters",
                                           systemImage: "line.3.horizontal.decrease.circ
                                           description: Text("Choose All to see everythi
                }
            }
        } else {
            List {
                ForEach(shown) { item in
                    NavigationLink(value: item) { ItemRow(item: item) }
                }
            }
        }
    }
}
```

The "no results" state is the "do I own this?" answer in its first form: search "Dune", see nothing, and a button offers to add "Dune" with the title already filled in.

`ContentUnavailableView.search(text:)` is the system's ready-made no-results view.

`catalogueIsEmpty` comes from the parent, because this view's own query is filtered and so cannot tell "nothing exists" from "nothing matches".

Step 3: The Parent Screen

`ListView` now owns the search field and the tag row, passes the text down, and keeps a plain unfiltered `@Query` of all items, used only to know whether the catalogue is empty and which tags are in use. (A second fetch of everything is fine at this size.) Replace the old body with:

```
struct ListView: View {
    @Query private var allItems: [Item]
    @Query(sort: \ItemType.sortOrder) private var types: [ItemType]
    @State private var model = ListViewModel()
    @State private var showAdd = false
    @State private var addTitle = ""

    // Tags that at least one item uses, in name order.
    private var tagsInUse: [Tag] {
        var seen = Set<String>()
        return allItems.flatMap(\.tags)
            .filter { seen.insert($0.key).inserted }
            .sorted { $0.name.localizedCaseInsensitiveCompare($1.name) == .orderedAscend
    }

    var body: some View {
        NavigationStack {
            VStack(spacing: 0) {
                TypeFilterRow(types: types, selected: $model.selectedTypeName)
                TagFilterRow(tags: tagsInUse, selectedKey: $model.selectedTagKey)
                ItemListContent(model: model, searchText: model.searchText,
                                catalogueIsEmpty: allItems.isEmpty) { title in
                    addTitle = title
                    showAdd = true
                }
            }
            .navigationTitle("My Catalogue")
            .searchable(text: $model.searchText, prompt: "Title or creator")
            .navigationDestination(for: Item.self) { item in DetailView(item: item) }
            .toolbar {
                ToolbarItem(placement: .primaryAction) {
                    Button("Add", systemImage: "plus") { addTitle = ""; showAdd = true }
                }
```

```
            }
            .sheet(isPresented: $showAdd) { ItemFormView(item: nil, initialTitle: addTit
        }
    }
}

struct TagFilterRow: View {
    let tags: [Tag]
    @Binding var selectedKey: String?

    var body: some View {
        if !tags.isEmpty {
            ScrollView(.horizontal, showsIndicators: false) {
                HStack(spacing: 8) {
                    FilterChip(title: "Any tag", isSelected: selectedKey == nil) { selec
                    ForEach(tags) { tag in
                        FilterChip(title: tag.name, isSelected: selectedKey == tag.key)
                    }
                }
                .padding(.horizontal)
                .padding(.bottom, 8)
            }
        }
    }
}
```

`.searchable` adds a search field that sits in the navigation bar and binds to the model's `searchText`. The tag row reuses Chapter 4's `FilterChip` and lists only tags that items really use, which also solves the orphan question left open in Chapter 7. If the chosen tag stops being used (its last item is deleted or retagged), the filter would silently hide everything: Exercise 2 asks you to handle that.

Step 4: "Do I Already Own This?" in the Form

The shared spec says a possible duplicate (same title and creator) is a **warning, never a block**: you may genuinely own two copies, or a book and its film. Add the starting title to the form and a warning footer. In `ItemFormView`:

```
// 1. New init parameter and an all-items query for the check:
@Query private var allItems: [Item]
```

```
init(item: Item?, initialTitle: String = "") {
    self.item = item
    var start = item.map(ItemDraft.init(item:)) ?? ItemDraft()
    if item == nil { start.title = initialTitle }
    _draft = State(initialValue: start)
}

// 2. The check: same folded title and creator, and not the item being edited.
private var duplicates: [Item] {
    let title = Tag.key(for: draft.title)
    let creator = Tag.key(for: draft.creator)
    guard !title.isEmpty else { return [] }
    return allItems.filter {
        $0 !== item && Tag.key(for: $0.title) == title && Tag.key(for: $0.creator) == cr
    }
}

// 3. In the title section's footer, after the error text:
if !duplicates.isEmpty {
    Label("You may already own this (\(duplicates.count) match). You can still save it."
          systemImage: "exclamationmark.triangle")
        .foregroundStyle(.orange)
}
```

It reuses `Tag.key(for:)` because that function already folds case and accents, and I am borrowing it for general strings; a clearer design would rename it to a shared `fold(_:)`. The warning appears as soon as both fields match an existing item, and Save is untouched, which is exactly the "never a block" rule. `$0 !== item` excludes the item being edited, so editing never warns about itself.

COMPARE WITH THE ANDROID COURSE

Personal Catalogue: Android (Kotlin & Compose) Chapter 8 used `debounce` and `flatMapLatest` to turn typing into queries without hammering the database. Here the equivalent is implicit: SwiftUI rebuilds the list view when the text changes, and the search happens in a fetch you never see. You gain simplicity and lose control: there is no debounce knob, so the sources' warning about re-fetching on every init is the thing to watch on a large store. The duplicate rule is the same in both courses, and the Android SQLite `NOCASE` limit is again avoided here by folding in code.

Challenges

EXERCISE 1

Make the changes, then add "Dune" by Frank Herbert and "Kind of Blue" by Miles Davis (tag the first "Sci-Fi"). Search "dune", "HERBERT", "blue" and "xyz". Write down what each shows. Then add "dune" by "frank herbert" and describe the warning.

View solution

EXERCISE 2

Select a tag in the filter row, then delete the only item carrying it. The list now shows "Nothing matches these filters" with no chip highlighted. Explain why, and fix it so the filter resets automatically.

View solution

EXERCISE 3

Test whether your search ignores accents: add an item titled "Café Society" and search "cafe", then add "Cafe Society" and "café" in turn. Record the results and the iOS version. Then say whether the duplicate check (which folds in code) and the search (which uses the database) agree, and why that could matter.

View solution

CHAPTER 8 QUICK REFERENCE

- `.searchable(text: $model.searchText, prompt: "...")` on the container inside the `NavigationStack`
- Dynamic query: a subview whose `init` sets `_items = Query(filter:sort:order:)` from the value passed in
- Case-insensitive match in a predicate: `localizedStandardContains`; never `uppercased()`; capture the search text in a local `let`
- Keep risky predicates out: filter the to-many `tags` in memory
- `ContentUnavailableView.search(text:)` for no results; offer "Add ..." with the typed text
- Duplicate = same folded title and creator, excluding the item being edited; a warning, never a block

NOT YET VERIFIED

Checked (one web search on 2026-11-19): that `localizedStandardContains` is the usual case-insensitive text match in a `#Predicate`, with a forum example; that it and `contains` on collections are listed among supported predicate operations by a third-party skill reference; that `uppercased()` in a predicate crashes at runtime; and that optional to-many relationships in predicates have been reported to crash, with making the relationship non-optional as a reported fix.

Not checked, from general knowledge: that `localizedStandardContains` ignores accents (the search did not say; Exercise 3 tests it); `.searchable` and where to attach it; the ternary choosing between two `#Predicate` values (if Xcode objects, use two initialisers or one predicate with the empty case folded in); `Query(filter:sort:order:)` with a `Predicate?`; `ContentUnavailableView.search(text:)`; that the rebuilt view re-queries on each text change; `$0 !== item` identity comparison for models; and all code, **none of which was compiled**. I did not consult Apple's reference pages.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Apple Developer Forums: SwiftData #Predicate case insensitive String comparison; Apple Developer Forums: How to handle predicate with optionals in SwiftData; Apple Developer Forums: SwiftData Predicates and .contains(where:); fatbobman: How to dynamically construct complex predicates for SwiftData; Swift Forums: Complex predicates in SwiftData; Michael Tsai: Optional values in SwiftData predicates; Predicate pitfalls (third-party skill reference). Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: iOS (Swift & SwiftUI) 9 adds polish: a light and dark appearance check, VoiceOver and Dynamic Type, and real error handling to replace the `print` calls. (The duplicate warning is already done in this chapter. Backup comes in Chapter 10.)

Personal Catalogue: iOS (Swift & SwiftUI)

Course 1 · Chapter 9 · Appearance, Accessibility and Errors

The app now does everything the spec asks, but three kinds of shortcut are still in it. Failures are `print`ed to a console nobody sees. Nothing has been checked in dark mode, with VoiceOver, or with the text size turned up. This chapter pays all three off.

> **A CORRECTION FIRST, AND WHAT WAS RUN**
>
> **The duplicate warning is already built.** The outline gives this chapter "Duplicates & Polish", but Chapter 8 absorbed the duplicate check because it belongs with search ("do I own this?"). I also wrote in Chapter 8 that this chapter would cover backup; that was wrong. **Backup is Chapter 10**, with the tests, and I have corrected Chapter 8's "Coming next" note. The takeaway for the course plan: check the outline row before announcing the next chapter.
>
> **Nothing here was compiled or run** (no Mac or Xcode). My search found third-party accessibility guides, not Apple's pages, and nothing on `ViewThatFits`, dark-mode colour rules or accessible error alerts; those are from memory.

Part 1: Light and Dark Appearance

SwiftUI adapts on its own as long as colours are *semantic*: `.primary`, `.secondary`, `.accentColor`, `Color(.secondarySystemBackground)`, and the system named colours like `.red` and `.orange`, which have light and dark variants. The risk is a fixed colour (`Color(red:green:blue:)`, a hex value, `.black`, `.white` used for a background). Audit what the app uses:

WHERE	COLOUR	VERDICT
`FilterChip` text on a selected chip	`Color.white` on `Color.accentColor`	Probably fine, but a fixed white on a variable accent: check contrast in both modes
`FilterChip` unselected	`.primary` on `secondarySystemBackground`	Semantic: adapts
`TagChip` background	`.tint.opacity(0.15)`	Adapts, but low contrast is possible; check it
Form errors, duplicate warning	`.red`, `.orange`	System colours: adapt. Orange text on a light background can be weak: check

WHERE	COLOUR	VERDICT
Row text	`.secondary`, `.tint`	Semantic: adapts

Verify instead of assuming: add a dark and a large-text preview next to the normal one for each screen. The scheme and the text size are both just environment values you can override in a preview:

```
#Preview("Dark") {
    ListView()
        .modelContainer(PreviewData.container)
        .preferredColorScheme(.dark)
}

#Preview("Large text") {
    ListView()
        .modelContainer(PreviewData.container)
        .environment(\.dynamicTypeSize, .accessibility3)
}
```

A preview is a quick check, not the final one: also toggle Dark Appearance and the text size on a simulator or device (the Xcode Environment Overrides button, or the Settings app), because some problems only show on the real screen. Fixing a weak spot is usually one line, for example choosing a darker text colour for the orange warning with `.foregroundStyle(.orange)` replaced by `.foregroundStyle(.primary)` next to the orange icon.

Part 2: VoiceOver

The sources agree on the main tools: a custom control needs a label; a group of views that mean one thing should be one VoiceOver element, with `.accessibilityElement(children: .combine)`; and a real `Button` gets focus handling and traits for free, which is why Chapter 4's chip is a `Button` and not an `onTapGesture`. Three changes:

```
// ItemRow: one spoken sentence per row instead of three separate fragments.
VStack(alignment: .leading, spacing: 4) {
    /* title, byline, type and tags as before */
}
.padding(.vertical, 2)
```

```
    .accessibilityElement(children: .combine)

    // FilterChip: say what tapping does.
    .accessibilityAddTraits(isSelected ? .isSelected : [])
    .accessibilityHint("Filters the list")

    // DetailView toolbar: icon-only buttons already carry their title as the label; confirm
```

Two things are decorative and should be hidden from VoiceOver if you add them (the sources mention this for icons): an icon that only repeats adjacent text gets `.accessibilityHidden(true)`. A row's tap action is the navigation link itself, which VoiceOver already announces as a button, so do *not* add a hint for that; hints that restate the obvious make the app slower to use, not better.

The honest way to check is to use it: turn VoiceOver on in a simulator or device (or open Xcode's Accessibility Inspector) and move through the list, the chips, the form and the delete dialog with your eyes shut or the screen dimmed. The sources are explicit that snapshot tests do not verify VoiceOver or Dynamic Type; only you using it does.

Part 3: Dynamic Type

Every piece of text in this app uses a *text style* (`.headline`, `.subheadline`, `.caption`), which scales with the user's size setting automatically. A hard-coded `.font(.system(size: 14))` would not, as one source warns; the app has none. What can still break is the *layout* around scaled text:

- **Chip rows** scroll horizontally, so large text only makes chips wider; nothing is cut off, but fewer are visible at once.
- **Row text**: the title has `.lineLimit(2)`, which at the largest sizes truncates long titles. Remove the limit (or raise it) for accessibility sizes, using `@Environment(\.dynamicTypeSize)` and `dynamicTypeSize.isAccessibilitySize` (a property I recall from memory; check Xcode accepts it).
- **The year field and fixed-size icons** that sit beside text should scale with it: use `@ScaledMetric` for any number that is a size, as the sources suggest for padding and image sizes.
- **Side-by-side layouts** such as the chip's label and remove button can overflow; `ViewThatFits` (not covered by my search, so from memory) lets a view choose a vertical

layout when the horizontal one does not fit.

Part 4: Real Error Handling

Chapters 3 to 6 each ended a `catch` with `print(...)` and a comment promising this chapter. Two situations, with different answers:

- **Save fails in the form.** The form is still on screen, so show an alert *there* and keep the sheet open so nothing the user typed is lost.
- **Delete fails after the detail screen has been dismissed** (Chapter 5's ordering). The screen that started it is gone, so the message needs somewhere else to live: a small app-wide **error centre** that the root view turns into an alert.

The form's alert is plain local state. Add to `ItemFormView`:

```
@State private var saveError: String?

// in save(): replace the print in the catch
} catch {
    saveError = error.localizedDescription        // the sheet stays open
}

// next to .toolbar on the Form
.alert("Couldn't save", isPresented: Binding(
    get: { saveError != nil },
    set: { if !$0 { saveError = nil } }
)) {
    Button("OK", role: .cancel) { }
} message: {
    Text(saveError ?? "")
}
```

A subtlety for `save()`: the repository inserts a new item *before* the save can fail, so a failed Save leaves a half-added object in the context. If the user fixes the problem and taps Save again, they could create a second one. The safe repository behaviour on failure is to undo what it inserted; the repository is the place for that, not the form. I did not rewrite it here, so Exercise 3 asks you to.

The error centre, for failures the current screen cannot show. Create `ErrorCenter.swift`:

```swift
import Observation

@Observable
final class ErrorCenter {
    var message: String?

    func report(_ error: Error, doing action: String) {
        message = "Couldn't \(action): \(error.localizedDescription)"
    }
}

// RootView: own it, share it, present it.
@State private var errors = ErrorCenter()
// on the root content:
.environment(errors)
.alert("Something went wrong", isPresented: Binding(
    get: { errors.message != nil },
    set: { if !$0 { errors.message = nil } }
)) {
    Button("OK", role: .cancel) { }
} message: {
    Text(errors.message ?? "")
}

// DetailView: read it from the environment, and use it in deleteItem's catch.
@Environment(ErrorCenter.self) private var errors
...
} catch {
    errors.report(error, doing: "delete the item")
}
```

`@Environment(ErrorCenter.self)` and `.environment(errors)` are the iOS 17 way to share an `@Observable` object (from memory; the search did not cover them). Any view that forgets to receive the object crashes at runtime, so previews that show `DetailView` must also supply `.environment(ErrorCenter())`. The same centre is where Chapter 10's import errors will go.

Not-found states. The Android course needed Loading and NotFound states because its detail screen loaded by id. This app passes the object, so those states do not exist; the nearest equivalent is Chapter 5's guard against showing a deleted model. There is also no "Loading" for lists, because `@Query` returns synchronously from the local store.

COMPARE WITH THE ANDROID COURSE

Personal Catalogue: Android (Kotlin & Compose) Chapter 9 handled errors with state in view models and a snackbar, and its accessibility work centred on TalkBack semantics and merged cards. The SwiftUI version has the same three jobs with fewer moving parts, because semantic colours and text styles are the default rather than something you set up. The price is the same on both platforms: the defaults get you most of the way, and the remaining problems only show when a person actually uses a screen reader or large text. Neither platform's tests will find them for you.

Challenges

EXERCISE 1

Run an audit: Dark Appearance, the largest accessibility text size, and VoiceOver (or the Accessibility Inspector) on the list, detail and form. Write down at least three real findings (something unclear, cut off, or low-contrast), the change you made for each, and the iOS version.

View solution

EXERCISE 2

Add the form's alert and the error centre. To test the form alert, make the repository's `save` temporarily throw. Record what the user sees, whether the sheet stays open, and whether their typing survives.

View solution

EXERCISE 3

Fix the half-inserted item problem: change `CatalogueRepository.save` so that when saving fails, anything it inserted is removed again and the error is rethrown. Explain why retrying Save must not create two items.

View solution

CHAPTER 9 QUICK REFERENCE

- Semantic colours only (`.primary`, `.secondary`, `.accentColor`, system backgrounds); preview with `.preferredColorScheme(.dark)`
- VoiceOver: `.accessibilityElement(children: .combine)`, `.accessibilityLabel`, `.accessibilityHint` (sparingly), `.accessibilityHidden(true)` for decoration
- Use `Button`, not `onTapGesture`, for tappable things
- Dynamic Type: text styles scale; fix layouts with `@ScaledMetric`, `ViewThatFits`, and lifting line limits at accessibility sizes; preview with `.environment(\.dynamicTypeSize, .accessibility3)`
- A failure on a visible screen: an `.alert` there, and keep the screen open. A failure after dismissal: an `@Observable` error centre in the environment, alerted from the root
- The repository should undo its own inserts when a save fails

NOT YET VERIFIED

Checked (one web search on 2026-11-19, third-party guides only): that `.accessibilityElement(children: .combine)` merges children into one VoiceOver element; that the main VoiceOver modifiers are label, hint, value and action; that `Button` gives focus handling and traits for free; that system text styles scale automatically and a fixed `.font(.system(size:))` does not; that `@ScaledMetric` scales a number with Dynamic Type; and that snapshot tests do not verify Dynamic Type or VoiceOver.

Not checked, from general knowledge: that the named system colours adapt (and the contrast judgements in the table, which you must test yourself); `ViewThatFits`; `dynamicTypeSize.isAccessibilitySize`; `.accessibilityHidden`; the `.alert` form with an item-less binding and a message closure; `@Environment(ErrorCenter.self)` and `.environment(errors)` for an `@Observable` class; `error.localizedDescription` being readable for SwiftData errors (it often is not); whether an alert can appear at the root while the detail screen is mid-dismissal; and all code, **none of which was compiled**. I did not consult Apple's accessibility pages.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): The Swift Kit: SwiftUI accessibility and VoiceOver guide; Swift Engineer: 5.13 Accessibility; SwiftUI: How to build Dynamic Type (iOS 17+); iOS Accessibility, SwiftUI and UIKit (third-party skill reference); Apple tech talk page (not opened). Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: iOS (Swift & SwiftUI) 10 is the last chapter: JSON backup and restore with `fileExporter` and `fileImporter` (reporting failures through this chapter's error centre), tests for the repository and the draft, and a roadmap with barcode scanning, compared with the Android course. The course PDF is built at the end.

Personal Catalogue: iOS (Swift & SwiftUI)

Course 1 · Chapter 10 · Backup, Tests and What Next

The last chapter adds the one feature the spec still lacks, **backup**: export the whole catalogue to a JSON file and import it back. It then adds **tests** for the logic worth protecting, and ends with a roadmap and an honest comparison of this app with the Android one.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Nothing in this chapter was compiled or run** (no Mac or Xcode here), and this is the chapter where that matters most, because a backup feature that has never been run is a backup you cannot trust. My search covered `fileExporter` with a `FileDocument`, security-scoped file access for imports, and VisionKit barcode scanning. It found **nothing on testing SwiftData** (XCTest, Swift Testing, in-memory containers); that part is from memory. Run the tests in this chapter before relying on any of the rest.

Part 1: The Backup Format

The file deliberately matches the Android app's format, so a backup from one app can be restored in the other: `version`, `exportedAt` and `items`, each item holding its type *by name* and its tags as names, and dates as **Unix milliseconds** (Android's choice; Swift's `Date` is converted at the edge). A human can read and repair the file.

```
import Foundation

struct BackupFile: Codable {
    static let currentVersion = 1
    var version = BackupFile.currentVersion
    var exportedAt: Int64
    var items: [BackupItem]
}

struct BackupItem: Codable {
    var title: String
    var creator = ""
    var formatDetail = ""
    var releaseYear: Int?
    var notes = ""
    var createdAt: Int64
```

```
        var type: String
        var tags: [String] = []

        init(title: String, creator: String = "", formatDetail: String = "", releaseYear: In
             notes: String = "", createdAt: Int64, type: String, tags: [String] = []) {
            self.title = title; self.creator = creator; self.formatDetail = formatDetail
            self.releaseYear = releaseYear; self.notes = notes
            self.createdAt = createdAt; self.type = type; self.tags = tags
        }

        // Swift's synthesised decoding rejects a missing key; hand-written, so absent optio
        init(from decoder: Decoder) throws {
            let c = try decoder.container(keyedBy: CodingKeys.self)
            title = try c.decode(String.self, forKey: .title)
            creator = try c.decodeIfPresent(String.self, forKey: .creator) ?? ""
            formatDetail = try c.decodeIfPresent(String.self, forKey: .formatDetail) ?? ""
            releaseYear = try c.decodeIfPresent(Int.self, forKey: .releaseYear)
            notes = try c.decodeIfPresent(String.self, forKey: .notes) ?? ""
            createdAt = try c.decode(Int64.self, forKey: .createdAt)
            type = try c.decode(String.self, forKey: .type)
            tags = try c.decodeIfPresent([String].self, forKey: .tags) ?? []
        }
    }

    extension Date {
        var unixMillis: Int64 { Int64((timeIntervalSince1970 * 1000).rounded()) }
        init(unixMillis: Int64) { self.init(timeIntervalSince1970: Double(unixMillis) / 1000
    }
```

Unknown extra keys (from a newer version) are ignored by `Codable` by default, matching the Android setting `ignoreUnknownKeys`.

Part 2: Export and Import

Three design decisions, mirrored from Android. **Import merges and never replaces:** an incoming item that matches an existing one on title, creator, type *and creation time* is skipped, so importing a file twice does not double the catalogue, while a deliberate second copy (different creation time) survives a round trip. **Import is all-or-nothing:** validate everything first (decode, version, every type name), only then insert, and save once. **An unknown type is an error**, not a guess.

All-or-nothing needs a repository change: the current `save(_:tagNames:)` inserts and saves each item. Split it into `stage(_:tagNames:)` (insert, find-or-create tags and assign them: everything the old function did *except* the final `context.save()`) and have `save` call `stage` and then save. Import uses `stage` for every item and saves once.

```
import Foundation
import SwiftData

struct ImportResult: Equatable { var added: Int; var skipped: Int }

enum BackupError: LocalizedError {
    case newerVersion
    case unknownType(String)
    var errorDescription: String? {
        switch self {
        case .newerVersion: return "This backup was made by a newer version of the app."
        case .unknownType(let t): return "Unknown item type \"\(t)\". This file does not
        }
    }
}

@MainActor
struct BackupService {
    let context: ModelContext

    func export() throws -> Data {
        let items = try context.fetch(FetchDescriptor<Item>(sortBy: [SortDescriptor(\.cr
        let file = BackupFile(
            exportedAt: Date.now.unixMillis,
            items: items.map {
                BackupItem(title: $0.title, creator: $0.creator, formatDetail: $0.formatI
                           releaseYear: $0.releaseYear, notes: $0.notes, createdAt: $0.c
                           type: $0.typeName, tags: $0.sortedTags.map(\.name))
            }
        )
        let encoder = JSONEncoder()
        encoder.outputFormatting = [.prettyPrinted, .sortedKeys]
        return try encoder.encode(file)
    }

    func importBackup(_ data: Data) throws -> ImportResult {
        // 1. Validate everything before touching the store.
        let file = try JSONDecoder().decode(BackupFile.self, from: data)
        guard file.version <= BackupFile.currentVersion else { throw BackupError.newerVe
```

```
        let repo = CatalogueRepository(context: context)
        try repo.ensureSeeded()
        let types = Dictionary(uniqueKeysWithValues: try repo.itemTypes().map { ($0.name
        for b in file.items where types[b.type.lowercased()] == nil {
            throw BackupError.unknownType(b.type)
        }

        // 2. Existing items, as a set of exact-match signatures.
        func signature(_ title: String, _ creator: String, _ type: String, _ created: In
            "\(title)\u{1F}\(creator)\u{1F}\(type.lowercased())\u{1F}\(created)"
        }
        var existing = Set(try context.fetch(FetchDescriptor<Item>()).map {
            signature($0.title, $0.creator, $0.typeName, $0.createdAt.unixMillis)
        })

        // 3. Stage every new item, then save once.
        var added = 0, skipped = 0
        for b in file.items {
            let title = b.title.trimmingCharacters(in: .whitespacesAndNewlines)
            let creator = b.creator.trimmingCharacters(in: .whitespacesAndNewlines)
            let sig = signature(title, creator, b.type, b.createdAt)
            if title.isEmpty || existing.contains(sig) { skipped += 1; continue }
            let item = Item(title: title, creator: creator, type: types[b.type.lowercase
            item.formatDetail = b.formatDetail.trimmingCharacters(in: .whitespacesAndNew
            item.releaseYear = b.releaseYear
            item.notes = b.notes.trimmingCharacters(in: .whitespacesAndNewlines)
            item.createdAt = Date(unixMillis: b.createdAt)
            try repo.stage(item, tagNames: b.tags)
            existing.insert(sig)
            added += 1
        }
        do {
            try context.save()
        } catch {
            context.rollback()                // nothing half-imported
            throw error
        }
        return ImportResult(added: added, skipped: skipped)
    }
}
```

I assumed `Item(title:creator:type:)` and the repository helpers keep their earlier shapes; adapt the calls to your files. `context.rollback()` is how I believe unsaved changes are

discarded; it is from memory and untested. The unit-separator character in the signature keeps "AB"+"C" from colliding with "A"+"BC". Note that the signature uses the item's *exact* title and creator where the duplicate warning folds case: an import should skip only a true copy.

Part 3: The Screen

A `FileDocument` wraps the bytes for the exporter. The sources show the pattern: a struct that supplies the data to write and, for a document type you also read, a way to read it back. Here the document is write-only, so it only needs to supply data:

```
import SwiftUI
import UniformTypeIdentifiers

struct BackupDocument: FileDocument {
    static var readableContentTypes: [UTType] { [.json] }
    var data: Data

    init(data: Data) { self.data = data }
    init(configuration: ReadConfiguration) throws {
        guard let bytes = configuration.file.regularFileContents else { throw CocoaError
        data = bytes
    }
    func fileWrapper(configuration: WriteConfiguration) throws -> FileWrapper {
        FileWrapper(regularFileWithContents: data)
    }
}

struct BackupView: View {
    @Environment(\.modelContext) private var context
    @Environment(\.dismiss) private var dismiss
    @Environment(ErrorCenter.self) private var errors
    @State private var exportDocument: BackupDocument?
    @State private var showExporter = false
    @State private var showImporter = false
    @State private var resultText: String?

    var body: some View {
        NavigationStack {
            Form {
                Section {
                    Button("Export backup...", systemImage: "square.and.arrow.up") { prep
                } footer: { Text("Saves every item as a JSON file you choose where to ke
```

```
                Section {
                    Button("Import backup...", systemImage: "square.and.arrow.down") { s
                } footer: { Text("Adds items from a backup. Existing items are never cha
            }
            .navigationTitle("Backup")
            .navigationBarTitleDisplayMode(.inline)
            .toolbar { ToolbarItem(placement: .confirmationAction) { Button("Done") { di
            .fileExporter(isPresented: $showExporter, document: exportDocument, contentT
                          defaultFilename: "catalogue-backup") { result in
                if case .failure(let error) = result { errors.report(error, doing: "expo
                else { resultText = "Backup saved." }
            }
            .fileImporter(isPresented: $showImporter, allowedContentTypes: [.json]) { re
                handleImport(result)
            }
            .alert("Backup", isPresented: Binding(get: { resultText != nil }, set: { if
                Button("OK", role: .cancel) { }
            } message: { Text(resultText ?? "") }
        }
    }

    private func prepareExport() {
        do {
            exportDocument = BackupDocument(data: try BackupService(context: context).ex
            showExporter = true          // both the document and the flag are needed be
        } catch {
            errors.report(error, doing: "prepare the backup")
        }
    }

    private func handleImport(_ result: Result<URL, Error>) {
        do {
            let url = try result.get()
            let scoped = url.startAccessingSecurityScopedResource()
            defer { if scoped { url.stopAccessingSecurityScopedResource() } }
            let data = try Data(contentsOf: url)
            let outcome = try BackupService(context: context).importBackup(data)
            resultText = "Added \(outcome.added) item(s), skipped \(outcome.skipped) alr
        } catch {
            errors.report(error, doing: "import the backup")
        }
    }
}
```

Why the security-scope calls: the sources say that a file chosen outside the app's own container cannot be read unless the URL's security scope is opened first, bracketed by `start...` and `stop...` with a `defer`; calling them on an in-container file is harmless. Reading happens inside the bracket, which is the whole point. Present `BackupView` as a sheet from a new toolbar button on `ListView` (for example a `Menu` with a "Backup" entry and the `externaldrive` symbol).

Part 4: Tests

Test the code that decides things, not the screens. Everything below is plain Swift or an *in-memory* container (`Persistence.makeContainer(inMemory: true)`, built in Chapter 3 for exactly this), so the tests run in milliseconds and touch no real data. I use XCTest because it exists in every Xcode; Apple's newer Swift Testing library (`import Testing`, `@Test`, `#expect`) is the alternative, and I did not verify either in my search.

```
import XCTest
import SwiftData
@testable import Catalogue          // your app target's module name

@MainActor
final class CatalogueTests: XCTestCase {

    private func makeContext() throws -> ModelContext {
        let container = try Persistence.makeContainer(inMemory: true)
        let context = container.mainContext
        try CatalogueRepository(context: context).ensureSeeded()
        return context
    }

    // ---- the draft: plain Swift, no database ----
    func testBlankTitleIsInvalid() {
        var d = ItemDraft(); d.typeName = "Book"; d.title = "   "
        XCTAssertTrue(d.titleError)
        XCTAssertFalse(d.isValid)
    }

    func testYearRules() {
        var d = ItemDraft(); d.typeName = "Book"; d.title = "Dune"
        d.yearText = "";     XCTAssertFalse(d.yearError)      // optional
        d.yearText = "1965"; XCTAssertFalse(d.yearError)
        d.yearText = "999";  XCTAssertTrue(d.yearError)
        d.yearText = "abcd"; XCTAssertTrue(d.yearError)
```

```
    }

    // ---- the repository, in memory ----
    func testTagsAreCaseInsensitiveAndShared() throws {
        let context = try makeContext()
        let repo = CatalogueRepository(context: context)
        let type = try repo.itemTypes()[0]
        try repo.save(Item(title: "Dune", creator: "", type: type), tagNames: ["Sci-Fi"]
        try repo.save(Item(title: "Alien", creator: "", type: type), tagNames: ["sci-fi"
        XCTAssertEqual(try context.fetch(FetchDescriptor<Tag>()).count, 1)
    }

    // ---- backup ----
    func testRoundTripAndMerge() throws {
        let source = try makeContext()
        let type = try CatalogueRepository(context: source).itemTypes()[0]
        try CatalogueRepository(context: source).save(Item(title: "Dune", creator: "Fran
        let data = try BackupService(context: source).export()

        let target = try makeContext()
        XCTAssertEqual(try BackupService(context: target).importBackup(data), ImportResu
        XCTAssertEqual(try BackupService(context: target).importBackup(data), ImportResu
    }

    func testUnknownTypeChangesNothing() throws {
        let context = try makeContext()
        let json = #"{"version":1,"exportedAt":0,"items":[{"title":"X","createdAt":1,"ty
        XCTAssertThrowsError(try BackupService(context: context).importBackup(Data(json.
        XCTAssertEqual(try context.fetch(FetchDescriptor<Item>()).count, 0)
    }
}
```

Two of these tests exercise things I flagged as unverified in earlier chapters (a find-before-insert on a unique key, and a clean import). If one fails, it has done its job: **read the failure before changing the test**. A test that you edit until it passes proves nothing.

Part 5: What Next, and Android Compared

Barcode scanning is the natural next feature: scan an ISBN or UPC and pre-fill the title. The sources describe VisionKit's `DataScannerViewController`: it is a UIKit view controller, so SwiftUI hosts it through a `UIViewControllerRepresentable`; you restrict it to barcode symbologies such

as EAN-13 (retail codes); it needs camera permission; and you should check `isSupported` and `isAvailable` before showing it. Turning a code into a title needs a lookup service (the open Open Library API for books, for example), and that is a network feature this offline app has so far avoided, so the privacy and failure behaviour must be designed first. Other ideas: cover images, a Home Screen widget, and iCloud sync. *Caution on sync, from memory only:* I recall that CloudKit-backed SwiftData does not allow unique constraints, which Chapter 2's tag `key` uses; verify before planning it.

CONCERN	ANDROID COURSE	IOS COURSE
Database	Room, explicit cross-reference table, `RESTRICT`	SwiftData, implicit join, `.deny`
Case-insensitive tags	`NOCASE` (ASCII only) + normalised column	Folded `key` (case and diacritics)
Live data to screen	Flow, ViewModel, `StateFlow`	`@Query`, `@Observable`
Navigation	Type-safe routes passing an id	`NavigationStack` passing the object
Form	State in a ViewModel	Draft struct in `@State`
Search	`debounce` + `flatMapLatest`	`.searchable` + re-initialised `@Query`
Backup	Storage Access Framework, same JSON	`fileExporter` / `fileImporter`, same JSON
Dependency injection	Hilt	The environment (`modelContext`, `ErrorCenter`)
Biggest risk	Boilerplate and wiring	Less control: model lifetimes and re-fetch timing

COMPARE WITH THE ANDROID COURSE

Personal Catalogue: Android (Kotlin & Compose) Chapter 10 ended with the same three parts: backup, tests, roadmap. The two apps now share a data model, a set of rules and a backup file, which was the purpose of building them as a pair. The honest summary: Android makes you assemble more, and you can see every piece; SwiftUI and SwiftData assemble more for you, and the problems that remain (deleting a model a screen still holds, predicates that crash, re-fetch timing) appear at runtime rather than as code you wrote. The Android course also ran its tests with a build tool I could not run here. Neither course's code was compiled, so the first job for either app is to open it in its IDE, build it, and fix what the compiler finds.

Challenges

EXERCISE 1

Split the repository's `save` into `stage` and `save`, add the backup files, `BackupView` and the toolbar entry. Export a catalogue of at least three items, open the file in a text editor and describe its structure, then import it into a fresh install and note the result message.

View solution

EXERCISE 2

Add the test file and run it. For each test that fails, report the failure message and whether the fault is in the code or in a wrong assumption of mine. Then add one test of your own for the `ListViewModel` type-and-tag filter.

View solution

EXERCISE 3

Cross-platform check: export from the iOS app and import into the Android app (or the reverse), using a catalogue with a diacritic in a title and a tag. Record what survives, what is skipped, and anything the two formats disagree on.

View solution

CHAPTER 10 QUICK REFERENCE

- Backup file: `{version, exportedAt, items[{title, creator, formatDetail, releaseYear, notes, createdAt (Unix ms), type (name), tags (names)}]}`, shared with the Android app
- Export: a `FileDocument` holding the bytes, `.fileExporter(isPresented:document:contentType:defaultFilename:)`; the document and the flag must both be set
- Import: `.fileImporter(isPresented:allowedContentTypes:)`, bracket the read with `startAccessingSecurityScopedResource()` / `stop...` in a `defer`

- Merge, never replace; skip exact matches (title, creator, type, creation time); validate everything first, stage all, save once, roll back on failure
- Tests: in-memory container, plain-Swift draft and view-model tests; read failures before editing tests
- Roadmap: VisionKit `DataScannerViewController` (check `isSupported` and `isAvailable`), cover images, widget, sync (check the unique-key limitation)

NOT YET VERIFIED

Checked (searches on 2026-11-19, tutorials and forum posts): that `fileExporter` takes a presentation binding and an optional document (both needed before the picker appears) and that a `FileDocument` wrapping data is the usual pattern; that `.json` is a standard import type; that files outside the app container need the security scope started and stopped, with calls on in-container files harmless; that a SwiftUI app hosts VisionKit's `DataScannerViewController` through a representable, restricts it to barcode symbologies, needs camera permission, and should check `isSupported` and `isAvailable`.
Not checked, from general knowledge: everything about testing SwiftData (the search found nothing): XCTest versus Swift Testing, `@MainActor` test classes, the `@testable import` module name, and in-memory containers in tests; `context.rollback()` discarding the staged inserts; the `stage` split of the repository, which depends on code from earlier chapters I am only recalling; the `fileExporter` completion-closure form used here (there are several overloads); that CloudKit-backed SwiftData forbids unique constraints; the Open Library suggestion; and all code, **none of which was compiled or run, including the tests**. I did not consult Apple's reference pages.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Use Your Loaf: SwiftUI importing and exporting files; Swift with Majid: File importing and exporting in SwiftUI; Apple Developer: fileExporter (one overload, not read in detail); Apple Developer Forums: Exporting a document with FileDocument; JSON import and export in SwiftUI (third-party blog); Medium: How to make a QR or barcode scanner in SwiftUI; Create with Swift: Reading QR codes and barcodes with the Vision framework; Kodeco: New scanning and text capabilities with VisionKit. Only titles and short summaries in the search results were used.

COURSE COMPLETE

That is the last chapter of *Personal Catalogue: iOS (Swift & SwiftUI)*. The honest status: the design and the code are written, researched where my searches could reach, and **entirely unbuilt**. Open a new Xcode project, work through the chapters in order, and treat every compiler error and failing test as

information about this course. Its Android counterpart, *Personal Catalogue: Android (Kotlin & Compose)*, shares the data model and the backup file.