

MOBILE CATALOGUE

Personal Catalogue: Android (Kotlin & Compose)

10 Chapters

Philip Osztromok

Generated with Claude

PERSONAL CATALOGUE: ANDROID (KOTLIN & COMPOSE)

Contents

Chapter 1	Project Setup	2
Chapter 2	Data Model	11
Chapter 3	Persistence Layer	20
Chapter 4	The List Screen	30
Chapter 5	The Detail Screen	40
Chapter 6	Add and Edit Form	50
Chapter 7	Tags	62
Chapter 8	Search and Filter	71
Chapter 9	Duplicates and Polish	82
Chapter 10	Backup, Tests and Next Steps	93

Personal Catalogue: Android (Kotlin & Compose)

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

This course builds a real app for your own phone: 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 Android app in **Kotlin** with **Jetpack Compose** for the screens and **Room** for the database. A sister course, *Personal Catalogue: iOS (Swift & SwiftUI)*, 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 has no Android Studio and no emulator. The steps and code come from my knowledge of Android Studio and from Google's developer documentation, and I checked only what the search results at the end of the chapter actually say. **The wizard's wording, the default `minSdk` and the exact version numbers change with every Android Studio release**, so wherever this chapter names one, trust what your own Android Studio shows over the text here, and treat the code as a starting point to compile, not as something already proven to build.
>
> **Updated 2026-11-21:** the wizard screens and the generated project described below were checked against screenshots and files from a real **Android Studio Rabbit 2** install (the Not Yet Verified box at the end says exactly what). The chapter's code has still not been compiled by me.

What We Are Building

The app has one job and a short list of features. Everything in this course is taken from the shared spec that both mobile courses use:

CHAPTER	FEATURE ADDED
1	The empty project, package layout, first screen on the emulator
2–3	The data model and the Room database, 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

CHAPTER	FEATURE ADDED
7	Tags: a managed, reusable list, with autocomplete
8	Search by title or creator, filter by tag
9	Duplicate warning, dark theme, accessibility, error states
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, and a later course can add them.

How This Differs from the Web Versions

If you built a web catalogue, most of the thinking carries over and only the machinery changes. This table shows where each web idea lands on Android:

	WEB CATALOGUE (DJANGO, PHP, EXPRESS)	THIS COURSE
Language	Python, PHP or JavaScript	Kotlin
Screens	HTML templates or React components	Composable functions (Jetpack Compose)
Database	PostgreSQL, MySQL, MongoDB or Firestore on a server	Room, a layer over SQLite, *on the phone*
Data entry	Django admin or an HTML form	A Compose 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 Android Emulator, or your own phone
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 simpler to run but the data lives on one device. That is why Chapter 10 builds a backup file: it is the Android equivalent of the database dump you would take from a web version.

Creating the Project

You need Android Studio installed. The *Android Development: Fundamentals* course and *Android Studio: The IDE Itself* cover installing it, so this chapter does not repeat that. To create the project, per Google's documentation:

1. Choose **File > New > New Project** (or **New Project** from the welcome screen).
2. In the template window choose **Empty Activity** and click Next. Older guides call this template *Empty Compose Activity (Material3)*; check the name your version shows. In Rabbit 2 the category is *Phone and large screens*; the XML-based templates are labelled *Views* (Basic, Empty and Responsive Views Activity), so Empty Activity is the Compose one.
3. Set **Name** to `Catalogue`. Set **Package name** to a package you own; this course uses `com.example.catalogue` in its examples, which you should replace with your own if you ever publish the app.
4. Choose a save location, and **keep spaces out of the whole path**: the wizard warns that whitespace in the save location can cause problems with the NDK tools (native code), which this app does not use but a later project might. There is no Language or Build configuration language field for this template: it generates Kotlin and the Kotlin DSL for you.
5. Pick a **minimum SDK** from the drop-down. The right choice is a trade-off (older versions reach more phones, newer ones give you more built-in features). In Rabbit 2 the wizard proposes **API 24 ("Nougat"; Android 7.0)** and shows that the app will run on approximately 99.5% of devices; a *Help me choose* link explains the trade-off. Accept the default your own Android Studio proposes and write the number down: later chapters mention it where it matters.
6. Click **Finish** and wait for the first Gradle sync to complete. The first sync downloads libraries and can take several minutes.

What the Wizard Generated

A new project is mostly Gradle configuration plus one activity. These are the parts you will actually touch in this course, in the **Project** view of the Project tool window:

```
Catalogue/
├── app/
│   ├── build.gradle.kts           <- this module's dependencies (Room goes here in Cha
│   └── src/
│       └── main/
│           ├── AndroidManifest.xml <- declares the app and its one activity
│           ├── java/com/example/catalogue/
│           │   ├── MainActivity.kt <- the entry point
│           │   └── ui/theme/       <- Color.kt, Theme.kt, Type.kt (Material 3 theme)
│           └── res/                <- strings, icons, themes
├── gradle/
│   └── libs.versions.toml          <- version catalog: one place for library versions
├── build.gradle.kts                <- project-level plugins
└── settings.gradle.kts             <- which modules the build includes
```

Your project also contains a few files this course does not touch: example tests in `app/src/test` and `app/src/androidTest`, a `keepRules` folder with a `rules.keep` file for the release-build shrinker, and two small XML files in `res/xml` about app backup. Leave them as generated.

Two files matter more than the rest. `MainActivity.kt` is where the app starts. `libs.versions.toml` is the **version catalog**: the generated build files read library names and versions from it, so when a later chapter adds Room you add a line there and a line in `app/build.gradle.kts`. Compose libraries are normally kept in step with each other through a **Compose BOM** (a "bill of materials" that fixes one compatible version for the whole Compose family), which is why the generated Compose dependencies do not each carry their own version number. Google's setup page lists Material 3 as `androidx.compose.material3:material3`; check your generated file for the exact lines and versions.

A Package Layout for the Whole Course

The wizard puts everything in one package. For a small app that is fine, but this one grows through ten chapters, so decide the layout now. The folders are by *responsibility*, not by screen type, which is the same split the web versions made between models, views and templates:

```
com.example.catalogue/
├── MainActivity.kt
├── data/            <- Room entities, DAOs, database, repository (Chapters 2-3)
├── ui/
│   ├── theme/       <- generated by the wizard
│   ├── list/        <- list screen + view model (Chapter 4)
│   ├── detail/      <- detail screen + view model (Chapter 5)
│   ├── form/        <- add/edit screen + view model (Chapter 6)
│   └── navigation/  <- the route table (Chapter 5)
└── backup/          <- JSON export and import (Chapter 10)
```

In Android Studio, switch the Project tool window to the **Android** view, right-click the `com.example.catalogue` package and choose **New > Package** for each of `data`, `ui.list`, `ui.detail`, `ui.form`, `ui.navigation` and `backup`. Empty packages are not shown until they hold a file, so each will appear once you add its first class in a later chapter. Creating them now is optional; the point is to know where each thing will go.

A First Screen of Our Own

Replace the generated greeting with a screen that says what the app is. The code below is a sketch of the usual shape of a generated Compose activity: an activity that calls `setContent`, which wraps a screen in the generated theme. Your generated file will differ in small ways (for example it calls `enableEdgeToEdge()` and includes a preview function); keep what the wizard gave you and change only the composable it displays. Two things then need doing by hand. First, **imports**: the wizard's import list does not include every name the sketch uses, so `Box`, `Alignment` and `MaterialTheme` will show in red until you click each one and press **Alt+Enter** and choose *Import* (or paste the sketch's import lines). Second, **the preview**: the generated `@Preview` function still calls `Greeting("Android")`, which no longer exists once you replace the greeting, so change that call to `CatalogueApp()`. Until both are done the file will not compile.

```
package com.example.catalogue

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
```

```
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import com.example.catalogue.ui.theme.CatalogueTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            CatalogueTheme {
                CatalogueApp()
            }
        }
    }
}

// The app's root composable. Later chapters replace the body with navigation.
@Composable
fun CatalogueApp() {
    Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
        Box(
            modifier = Modifier
                .fillMaxSize()
                .padding(innerPadding),
            contentAlignment = Alignment.Center
        ) {
            Text(
                text = "My Catalogue",
                style = MaterialTheme.typography.headlineMedium
            )
        }
    }
}
```

Three ideas in that file carry the whole course. A **composable** is a Kotlin function marked `@Composable` that *describes* some UI; Compose calls it again whenever the data it shows changes. A **Scaffold** is Material 3's standard screen frame (later it holds the top bar and a "+" button) and hands your content an `innerPadding` so that it does not sit underneath system bars. A **Modifier** is how you size, pad and align a composable: you chain them, and the order matters (here, fill the screen, then pad it).

Running It

1. Open **Tools > Device Manager** and create a virtual device (a recent Pixel phone image is a good default). Download the system image if Android Studio offers to.
2. Choose the device from the toolbar drop-down and press **Run** (the green triangle). The first build is slow; later ones are much faster.
3. When the emulator shows *My Catalogue* in the middle of the screen, you have a working toolchain.

To run on your own phone instead, turn on **Developer options** and **USB debugging** on the phone, connect it by cable, and it appears in the same drop-down. If the build fails, the **Build** tool window shows the first error at the top; fix that one first, since later errors often cascade from it.

> **COMPARE WITH THE IOS COURSE**
>
> In *Personal Catalogue: iOS (Swift & SwiftUI)* Chapter 1 does the same job: a new Xcode project, a folder layout, and a first screen. The mapping is direct: a composable function is a SwiftUI `View`, a `Scaffold` is roughly a `NavigationStack` with a toolbar, and the Android Emulator is the iOS Simulator. The two apps will end up with the same screens and the same data model.

Challenges

EXERCISE 1

Create the project exactly as described and run it on the emulator. Write down the **Name**, **Package name**, **minimum SDK** and the name of the template your Android Studio showed. Then find `minSdk` in the generated `app/build.gradle.kts` (or the version catalog) and confirm it matches the number you chose.

View solution

EXERCISE 2

Change the screen so that it shows the app name in the centre and, underneath it, a second, smaller line: "*Books, CDs, DVDs and Blu-rays*". You will need a `Column` inside the `Box`. Run it and say which `Modifier` or alignment 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 Android 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: Kotlin, Jetpack Compose with Material 3, Room, ViewModel with StateFlow, Navigation Compose
- New project: File > New > New Project > Empty Activity (Compose); no spaces in the save location; Kotlin and Kotlin DSL are automatic; accept the default minimum SDK (API 24 in Rabbit 2) and record it
- Version catalog: `gradle/libs.versions.toml`; app dependencies: `app/build.gradle.kts`; Compose versions are kept together by a BOM
- Package layout: `data`, `ui.list`, `ui.detail`, `ui.form`, `ui.navigation`, `backup`, plus the generated `ui.theme`
- A composable is a `@Composable` function that describes UI; `Scaffold` is the screen frame; `Modifier` chains size, padding and alignment
- Run: Device Manager, create a virtual device, press Run; or use a phone with USB debugging

NOT YET VERIFIED

Checked (one web search on 2026-11-19): Google's steps to start a project (File > New > New Project, then Empty Activity); that Material 3 is the artifact `androidx.compose.material3:material3`; that current docs call the template "Empty Activity" while older guides say "Empty Compose Activity (Material3)"; and that Compose uses a BOM. **The search did not confirm the wizard's default minimum SDK**; the screenshots below did.
Checked on a real Android Studio Rabbit 2 (2026-11-21, from the reader's screenshots and project files): the template list and the *Views* labels; the two wizard pages and their fields (Name, Package name, Save location, Minimum SDK, no Language or build configuration field for Empty Activity); the default minimum SDK of API 24 and the whitespace warning; that the generated project

uses the Kotlin DSL and a version catalog with a Compose BOM; the generated file tree (plus the extra test, keepRules and backup files); that `MainActivity.kt` calls `enableEdgeToEdge()` and generates a preview that calls `Greeting`; and that Device Manager is under `Tools` and in the right-hand tool window bar.

Not checked, from general knowledge: that the sketch compiles once the imports and the preview are fixed (it was never built; `CatalogueTheme` is derived from the project name); the USB debugging steps; the package layout (a design choice, not a platform rule); and the comparison with the iOS course. Compile it, and correct anything your Android Studio does differently.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Android Developers, Set up the Compose Compiler Gradle plugin; Android Developers, Compose setup. Only the titles and short summaries in the search results were used.

COMING NEXT

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

Personal Catalogue: Android (Kotlin & Compose)

Course 1 · Chapter 2 · Entities, Relations and Indices

Chapter 1 left you with an empty project and a "My Catalogue" screen. This chapter writes the **shape of the data**: four Kotlin classes that Room will turn into four SQLite tables. There is no database object, no queries and no screen yet; those come in Chapter 3. Getting the model right first matters because every later chapter reads or writes these classes, and changing a table after the app is installed means writing a migration.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Nothing in this chapter was run or compiled**, as in Chapter 1: this machine has no Android Studio. The Room pattern for a many-to-many relation was checked against Google's documentation (see the boxes at the end); the rest is written from general knowledge of Room and Kotlin. **Room's annotation processor checks entities when you build**, so your first build is the real test: read its error messages, they are usually precise.

From Django Models to Room Entities

If you built a web catalogue, you already made these decisions once. An Android **entity** is a Kotlin `data class` marked `@Entity`; Room creates one table per entity and one column per property.

CONCEPT	DJANGO	ROOM
A table	`class Item(models.Model)`	`@Entity data class Item`
Primary key	Automatic `id`	`@PrimaryKey(autoGenerate = true) val id: Long = 0`
Optional field	`null=True, blank=True`	A nullable type: `val releaseYear: Int?`
One-to-many	`ForeignKey(ItemType)`	An `itemTypeId` column plus `@ForeignKey`
Many-to-many	`ManyToManyField(Tag)` (the join table is hidden)	An explicit cross-reference entity you write yourself
Unique value	`unique=True`	An `@Index(unique = true)`

CONCEPT	DJANGO	ROOM
Creating the tables	`makemigrations` and `migrate`	Room generates them from the entities when the database is built (Chapter 3)

The one real difference is the many-to-many row. Django hides the join table; Room makes you write it, and that explicit class is the most important thing to understand in this chapter.

The Four Entities

Create a file for each entity in the `data` package you planned in Chapter 1 (right-click it, **New > Kotlin Class/File**). Start with the two simple ones.

ItemType

A fixed lookup list (Book, CD, DVD, Blu-ray) that Chapter 3 seeds on first launch. Its name must be unique, so it gets a unique index.

```
package com.example.catalogue.data

import androidx.room.Entity
import androidx.room.Index
import androidx.room.PrimaryKey

@Entity(
    tableName = "item_type",
    indices = [Index(value = ["name"], unique = true)]
)
data class ItemType(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    val name: String
)
```

Tag

A tag is just a name, but the spec says "*Web Development*" and "*web development*" must not both exist. A unique index alone would allow both, because SQLite compares text case-sensitively by default. The usual fix is to give the column a case-insensitive collation, so the unique index compares names without regard to case.

```
package com.example.catalogue.data

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.Index
import androidx.room.PrimaryKey

@Entity(
    tableName = "tag",
    indices = [Index(value = ["name"], unique = true)]
)
data class Tag(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    @ColumnInfo(collate = ColumnInfo.NOCASE) val name: String
)
```

Note that `NOCASE` in SQLite folds only the 26 basic English letters, so it handles "Sci-Fi" and "sci-fi" but not accented letters such as "É". For a personal catalogue that is an acceptable limit; Chapter 7 also trims and normalises names before saving, so the database rule is a safety net rather than the only defence.

Item

The main entity. It points at its type with a foreign key, and everything else is plain data. Two choices are worth explaining. The creation time is a `Long` (milliseconds since 1970), which Room stores natively and which needs no converter. And the optional year is `Int?`, which Room stores as a nullable column.

```
package com.example.catalogue.data

import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.Index
import androidx.room.PrimaryKey

@Entity(
    tableName = "item",
    foreignKeys = [
        ForeignKey(
            entity = ItemType::class,
            parentColumns = ["id"],
```

```
            childColumns = ["itemTypeId"],
            onDelete = ForeignKey.RESTRICT
        )
    ],
    indices = [
        Index(value = ["itemTypeId"]),
        Index(value = ["title"])
    ]
)
data class Item(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    val title: String,
    val creator: String = "",
    val formatDetail: String = "",
    val releaseYear: Int? = null,
    val notes: String = "",
    val createdAt: Long = System.currentTimeMillis(),
    val itemTypeId: Long
)
```

`onDelete = RESTRICT` means SQLite refuses to delete an item type while items still use it. That is the right rule here, since types are a fixed list the user never deletes; it turns a bug in your own code into an error instead of silent data loss. The two indices serve later chapters: `itemTypeId` makes the list's type filter fast (and Room's documentation recommends indexing a foreign-key child column), and `title` helps the search screen.

ItemTagCrossRef

A tag can be on many items and an item can have many tags, so neither table can hold the link. A third table does: one row per (item, tag) pair. Per Google's guidance, this **cross-reference entity** has a *composite primary key* made of both ids, so the same tag cannot be attached to the same item twice.

```
package com.example.catalogue.data

import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.Index

@Entity(
    tableName = "item_tag",
    primaryKeys = ["itemId", "tagId"],
```

```
        foreignKeys = [
            ForeignKey(
                entity = Item::class,
                parentColumns = ["id"],
                childColumns = ["itemId"],
                onDelete = ForeignKey.CASCADE
            ),
            ForeignKey(
                entity = Tag::class,
                parentColumns = ["id"],
                childColumns = ["tagId"],
                onDelete = ForeignKey.CASCADE
            )
        ],
        indices = [Index(value = ["tagId"])]
    )
    data class ItemTagCrossRef(
        val itemId: Long,
        val tagId: Long
    )
```

Here `CASCADE` is deliberate: deleting an item removes its links (but never the tags themselves), and deleting a tag removes it from every item. The composite key already gives an index that starts with `itemId`, so only `tagId` needs its own index, for the "which items have this tag?" query in Chapter 8.

RELATIONSHIP	ON DELETE	WHY
Item → ItemType	`RESTRICT`	Types are a fixed list; deleting one that is in use is a bug
Link → Item	`CASCADE`	Deleting an item should remove its tag links
Link → Tag	`CASCADE`	Deleting a tag should detach it everywhere

Loading an Item with Its Type and Tags

The list and detail screens want an item *together with* its type name and its tags, not four separate lookups. Room models that with a plain (non-entity) class that embeds the item and declares its relations. The many-to-many one names the cross-reference through `Junction`:

```kotlin
package com.example.catalogue.data

import androidx.room.Embedded
import androidx.room.Junction
import androidx.room.Relation

data class ItemWithDetails(
    @Embedded val item: Item,

    @Relation(
        parentColumn = "itemTypeId",
        entityColumn = "id"
    )
    val type: ItemType,

    @Relation(
        parentColumn = "id",
        entityColumn = "id",
        associateBy = Junction(
            value = ItemTagCrossRef::class,
            parentColumn = "itemId",
            entityColumn = "tagId"
        )
    )
    val tags: List<Tag>
)
```

Read it as three sentences. *The item is embedded. Its type is the* `ItemType` *whose* `id` *equals the item's* `itemTypeId`*. Its tags are the* `Tag`*s reached by going through* `ItemTagCrossRef`*, from this item's* `id` *to each link's* `itemId`*, then from* `tagId` *to the tag's* `id`. Chapter 3 writes the DAO query that returns this class; the DAO method needs `@Transaction` because Room runs several queries to fill the relations.

COMPARE WITH THE IOS COURSE

In *Personal Catalogue: iOS (Swift & SwiftUI)* Chapter 2, SwiftData models are classes marked `@Model`, and the relationship is a property such as `var tags: [Tag]`: there is no cross-reference class to write. Android's data model is longer because Room is a thin layer over SQL tables, while SwiftData hides the join table, as Django does. The fields, types and rules (unique tag names, no deleting a type in use) are identical.

Common Mistakes

- **Forgetting the foreign-key index.** Room warns at build time when a child column of a foreign key has no index. Add the index rather than suppressing the warning.
- **Mismatched column names.** `childColumns`, `primaryKeys` and `Index(value = ...)` use the *column* names (the property names unless you change them with `@ColumnInfo`), spelled exactly.
- **Storing a tag list inside the item.** A comma-separated string works until the first rename. A link table is the correct, queryable design.
- **Making `id` non-default.** With `autoGenerate = true` the property needs a default of 0 so you can create an item before it has been saved.

Challenges

EXERCISE 1

Create the five files (`ItemType.kt`, `Tag.kt`, `Item.kt`, `ItemTagCrossRef.kt`, `ItemWithDetails.kt`) in the `data` package. You cannot build a database yet, so check them by building the project (**Build > Make Project**) only if Room is already in your Gradle files; otherwise just confirm the files have no red underlines once you add the Room dependency named in Chapter 3. Write down what each import is for.

View solution

EXERCISE 2

The spec says a duplicate warning compares title and creator, case-insensitively (Chapter 9). Decide whether to enforce that with a unique index on `Item` or to do it in code, and justify the choice. Remember that the spec calls it *a warning, never a block*.

View solution

EXERCISE 3

Sketch, on paper or in a text file, the rows of all four tables after you add a book "Dune" tagged *sci-fi* and *classic*, and a DVD "Alien" tagged *sci-fi*. Then say which rows disappear if you delete the tag *sci-fi*,

and which if you delete "Dune".

View solution

CHAPTER 2 QUICK REFERENCE

- Entity: `@Entity data class`; one table per entity, one column per property; nullable property = nullable column
- Key: `@PrimaryKey(autoGenerate = true) val id: Long = 0`; composite key: `@Entity(primaryKeys = ["a", "b"])`
- Foreign key: `ForeignKey(entity, parentColumns, childColumns, onDelete)`; index the child column
- Unique value: `Index(value = ["name"], unique = true)`; case-insensitive text: `@ColumnInfo(collate = ColumnInfo.NOCASE)`
- Many-to-many: an explicit cross-reference entity with a composite key and two foreign keys
- Reading relations: a non-entity class with `@Embedded` and `@Relation` (`associateBy = Junction(...)` for many-to-many)
- Delete rules used here: `RESTRICT` on Item → ItemType, `CASCADE` on both link foreign keys

NOT YET VERIFIED

Checked (one web search on 2026-11-19): that Google's documentation models many-to-many with a third cross-reference class using a composite `primaryKeys` and indices; that the reading class uses `@Embedded` plus `@Relation` with `associateBy = Junction(...)`; that those queries are run in a `@Transaction`; and that the `ForeignKey` reference says the parent column must be a primary key or unique, with an index on the child table recommended.

Not checked, from general knowledge: that Google's own example does *not* add `foreignKeys` to the cross-reference (the search did not show one; I added them as consistent with the `ForeignKey` reference); the `RESTRICT` and `CASCADE` behaviour described; `ColumnInfo.NOCASE` and its English-letters-only limit; the exact `Junction` parameter names; Room's build-time warning about unindexed foreign keys; and every import and every line of code, none of which was compiled.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Android Developers, Define and query many-to-many relationships; Android Developers, Choose relationship types between objects; Junction API reference; ForeignKey API reference. Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: Android (Kotlin & Compose) 3 adds the Room dependencies, the DAOs, the `AppDatabase` class, seeding the four item types on first launch, the repository, and light Hilt wiring.

Personal Catalogue: Android (Kotlin & Compose)

Course 1 · Chapter 3 · DAOs, Database, Repository and Hilt

Chapter 2 described the data; this chapter makes it real. You will add Room to the project, write the **DAOs** (the interfaces that hold your queries), build the **database class**, seed the four item types, put a **repository** in front of it all, and wire the pieces together with **Hilt**. 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 A VERSION WARNING**
>
> **Nothing here was compiled or run**; there is no Android Studio on this machine. More importantly, my search found that **Room now has two generations**. Room 2.x (the `androidx.room` packages, which Chapter 2 and this chapter use) and the newer **Room 3.0**, whose packages are `androidx.room3`, which requires KSP and changes some APIs. A new project today might be offered either. This course is written against Room 2.x; if you choose Room 3, the ideas are the same but every `androidx.room` import, the artifact names and some signatures change, and you should follow Google's migration page (linked at the end).

Where This Layer Sits

LAYER	ANDROID CLASS	WEB CATALOGUE EQUIVALENT
Tables	Entities (Chapter 2)	Django models
Queries	DAO interfaces	ORM queries, `Item.objects.filter()`
Connection	`RoomDatabase` subclass	The database connection and settings
Business rules	Repository class	A service or model-manager layer
Wiring	Hilt modules	Dependency setup in the framework (`settings.py`, app factory)

The spec's rule is **one repository class between the UI and the database**. Screens (Chapter 4 onward) will talk only to the repository, never to a DAO, which keeps the rules in one place and makes the view models testable in Chapter 10.

Adding Dependencies

Room needs a runtime library, an annotation processor, and (per Google's setup page) **KSP** to run that processor. `room-ktx` adds coroutine support, including the `withTransaction` function used below. Hilt needs its own plugin, runtime and compiler, plus a Compose-aware helper for view models. Add them through the version catalog you met in Chapter 1. **The version numbers are placeholders: copy the current ones from the release pages**, and make the KSP version match your Kotlin version, as KSP releases are tied to Kotlin releases.

```
# gradle/libs.versions.toml
[versions]
room = "x.y.z"          # Room 2.x line; see the Room release page
hilt = "x.y.z"
ksp = "x.y.z-x.y.z"    # must match your Kotlin version
hiltNavigationCompose = "x.y.z"

[libraries]
androidx-room-runtime = { module = "androidx.room:room-runtime", version.ref = "room" }
androidx-room-ktx = { module = "androidx.room:room-ktx", version.ref = "room" }
androidx-room-compiler = { module = "androidx.room:room-compiler", version.ref = "room"
hilt-android = { module = "com.google.dagger:hilt-android", version.ref = "hilt" }
hilt-compiler = { module = "com.google.dagger:hilt-android-compiler", version.ref = "hil
androidx-hilt-navigation-compose = { module = "androidx.hilt:hilt-navigation-compose", v

[plugins]
ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
hilt = { id = "com.google.dagger.hilt.android", version.ref = "hilt" }
```

```
// app/build.gradle.kts  (add inside the existing blocks)
plugins {
    // ...the generated plugins stay...
    alias(libs.plugins.ksp)
    alias(libs.plugins.hilt)
}

dependencies {
    // ...the generated dependencies stay...
    implementation(libs.androidx.room.runtime)
    implementation(libs.androidx.room.ktx)
    ksp(libs.androidx.room.compiler)

    implementation(libs.hilt.android)
    ksp(libs.hilt.compiler)
```

```
    implementation(libs.androidx.hilt.navigation.compose)
}
```

The two plugins must also be declared (with `apply false`) in the project-level `build.gradle.kts`, the same way the wizard declared the others. Then use **File > Sync Project with Gradle Files**. The red underlines in Chapter 2's files should vanish; if one remains, read the sync error first.

The DAOs

A **DAO** (data access object) is an interface whose methods carry SQL or an annotation such as `@Insert`. Room writes the implementation for you at build time. Methods that read return a `Flow`, which re-emits whenever the underlying tables change; that is what will keep the list screen up to date with no manual refreshing. Methods that write are `suspend` functions, so they run off the main thread.

```
package com.example.catalogue.data

import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Transaction
import androidx.room.Update
import kotlinx.coroutines.flow.Flow

@Dao
interface ItemTypeDao {
    @Query("SELECT * FROM item_type ORDER BY id")
    fun observeAll(): Flow<List<ItemType>>

    @Query("SELECT COUNT(*) FROM item_type")
    suspend fun count(): Int

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insertAll(types: List<ItemType>)
}

@Dao
```

```kotlin
interface TagDao {
    @Query("SELECT * FROM tag ORDER BY name")
    fun observeAll(): Flow<List<Tag>>

    @Query("SELECT * FROM tag WHERE name = :name LIMIT 1")
    suspend fun findByName(name: String): Tag?

    // Returns the new row id, or -1 if the name already exists (ignored by the unique i
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(tag: Tag): Long
}

@Dao
interface ItemDao {
    @Transaction
    @Query("SELECT * FROM item ORDER BY createdAt DESC")
    fun observeAll(): Flow<List<ItemWithDetails>>

    @Transaction
    @Query("SELECT * FROM item WHERE id = :id")
    fun observeById(id: Long): Flow<ItemWithDetails?>

    @Insert
    suspend fun insert(item: Item): Long

    @Update
    suspend fun update(item: Item)

    @Delete
    suspend fun delete(item: Item)
}

@Dao
interface ItemTagDao {
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insertAll(refs: List<ItemTagCrossRef>)

    @Query("DELETE FROM item_tag WHERE itemId = :itemId")
    suspend fun deleteForItem(itemId: Long)
}
```

Put them in `ItemDao.kt`, `TagDao.kt`, `ItemTypeDao.kt` and `ItemTagDao.kt` (Kotlin allows several per file, but one per file is easier to find). Notice `@Transaction` on the two methods that return `ItemWithDetails`: as Chapter 2 said, Room runs several queries to fill the type and tags,

and the annotation makes them one consistent read. Notice also that `findByName` compares `name = :name` with no case handling: because the `Tag.name` column was declared `NOCASE` in Chapter 2, SQLite should apply that collation to the comparison, so a search for "SCI-FI" finds "sci-fi".

The Database Class

The database class lists every entity, declares a version, and exposes the DAOs. It is abstract: Room generates the concrete class.

```
package com.example.catalogue.data

import androidx.room.Database
import androidx.room.RoomDatabase

@Database(
    entities = [ItemType::class, Item::class, Tag::class, ItemTagCrossRef::class],
    version = 1,
    exportSchema = true
)
abstract class AppDatabase : RoomDatabase() {
    abstract fun itemTypeDao(): ItemTypeDao
    abstract fun tagDao(): TagDao
    abstract fun itemDao(): ItemDao
    abstract fun itemTagDao(): ItemTagDao
}
```

`version = 1` matters later: when you change an entity after the app is installed, you raise the version and supply a **migration**. While developing, uninstalling the app from the emulator is an easy way to start from a clean database. `exportSchema = true` asks Room to write a JSON description of each version, which migrations and tests use; Room may warn at build time if you have not told it where to write the files (a `room.schemaLocation` setting in the KSP configuration), so set one if it does.

Hilt Wiring

Something must build the one database and hand it to whoever needs it. You could write a hand-made singleton, but this course follows Android Development: Architecture & Data and uses Hilt, lightly: three small pieces.

```
// CatalogueApplication.kt  (package com.example.catalogue)
import android.app.Application
import dagger.hilt.android.HiltAndroidApp

@HiltAndroidApp
class CatalogueApplication : Application()

// MainActivity.kt: add one annotation above the existing class line
import dagger.hilt.android.AndroidEntryPoint

@AndroidEntryPoint
class MainActivity : ComponentActivity() { /* unchanged */ }

// data/DatabaseModule.kt
package com.example.catalogue.data

import android.content.Context
import androidx.room.Room
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.android.qualifiers.ApplicationContext
import dagger.hilt.components.SingletonComponent
import javax.inject.Singleton

@Module
@InstallIn(SingletonComponent::class)
object DatabaseModule {

    @Provides
    @Singleton
    fun provideDatabase(@ApplicationContext context: Context): AppDatabase =
        Room.databaseBuilder(context, AppDatabase::class.java, "catalogue.db").build()

    @Provides fun provideItemTypeDao(db: AppDatabase): ItemTypeDao = db.itemTypeDao()
    @Provides fun provideTagDao(db: AppDatabase): TagDao = db.tagDao()
    @Provides fun provideItemDao(db: AppDatabase): ItemDao = db.itemDao()
    @Provides fun provideItemTagDao(db: AppDatabase): ItemTagDao = db.itemTagDao()
}
```

Register the application class in `AndroidManifest.xml` by adding `android:name=".CatalogueApplication"` to the `<application>` tag. Without that line Hilt never starts and the build or first launch fails with a Hilt error. `@Singleton` on `provideDatabase` is essential: opening several instances of one database file is a bug.

The Repository

The repository holds the logic that is more than a single query: saving an item *and* its tag links as one unit, and making sure the four item types exist. `@Inject constructor` lets Hilt build it, and `@Singleton` keeps one copy.

```
package com.example.catalogue.data

import androidx.room.withTransaction
import kotlinx.coroutines.flow.Flow
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class CatalogueRepository @Inject constructor(
    private val db: AppDatabase,
    private val itemTypeDao: ItemTypeDao,
    private val tagDao: TagDao,
    private val itemDao: ItemDao,
    private val itemTagDao: ItemTagDao
) {
    val itemTypes: Flow<List<ItemType>> = itemTypeDao.observeAll()
    val tags: Flow<List<Tag>> = tagDao.observeAll()
    val items: Flow<List<ItemWithDetails>> = itemDao.observeAll()

    fun observeItem(id: Long): Flow<ItemWithDetails?> = itemDao.observeById(id)

    /** Call once at startup. Safe to call every launch: it only inserts when the table
    suspend fun ensureSeeded() {
        if (itemTypeDao.count() == 0) {
            itemTypeDao.insertAll(
                listOf("Book", "CD", "DVD", "Blu-ray").map { ItemType(name = it) }
            )
        }
    }
```

```
    /** Saves a new item (id == 0) or updates an existing one, and replaces its tag links. */
    suspend fun saveItem(item: Item, tagIds: Set<Long>): Long = db.withTransaction {
        val id = if (item.id == 0L) {
            itemDao.insert(item)
        } else {
            itemDao.update(item)
            item.id
        }
        itemTagDao.deleteForItem(id)
        itemTagDao.insertAll(tagIds.map { ItemTagCrossRef(itemId = id, tagId = it) })
        id
    }

    suspend fun deleteItem(item: Item) = itemDao.delete(item)

    /** Returns the existing tag with this name (any case), or creates it. */
    suspend fun getOrCreateTag(rawName: String): Tag {
        val name = rawName.trim()
        tagDao.findByName(name)?.let { return it }
        val newId = tagDao.insert(Tag(name = name))
        return Tag(id = newId, name = name)
    }
}
```

The seeding function is called from the activity or a view model in Chapter 4: `ensureSeeded()` checks the count and inserts only when the table is empty, so it is safe on every launch. A `RoomDatabase.Callback.onCreate` is the other common place to seed, but its method signature differs between Room versions (the older `SupportSQLiteDatabase`, the newer `SQLiteConnection`, and `suspend` functions in Room 3), whereas this repository approach does not depend on any of them.

A subtle point in `getOrCreateTag`: if two tags are created at once, the second `insert` returns `-1` because of the unique index, and the code above would then return a tag with a wrong id. Chapter 7 tightens this by re-reading the tag after an ignored insert; for now the form creates tags one at a time, so it cannot happen.

COMPARE WITH THE IOS COURSE

In *Personal Catalogue: iOS (Swift & SwiftUI)* Chapter 3 the same layer is smaller: SwiftData's `ModelContainer` replaces the database class and the DAOs, queries come from `@Query` in the views, and no dependency-injection library is needed. The repository still exists to hold "save with tags" and

the seeding. Android pays for flexibility with more setup; SwiftUI pays for brevity with less control over where queries live.

Challenges

EXERCISE 1

Add the dependencies and plugins, create `CatalogueApplication`, annotate `MainActivity`, register the application in the manifest, and add the database class, DAOs, module and repository. Build the project. List every error you got and which of the setup steps each pointed to.

View solution

EXERCISE 2

Add a DAO query that returns how many items have a given tag id, as a `Flow<Int>`, and expose it from the repository. Say which index from Chapter 2 makes it quick.

View solution

EXERCISE 3

Without writing code, explain what happens if `saveItem` is written *without* `db.withTransaction` and the app is killed between the item insert and the tag insert. Then explain why that cannot leave an orphan row when the transaction is used.

View solution

CHAPTER 3 QUICK REFERENCE

- Room needs `room-runtime`, `room-ktx` and the compiler through `ksp`; Hilt needs its plugin, `hilt-android` and its compiler through `ksp`. This course uses Room 2.x (`androidx.room`); Room 3 uses `androidx.room3`
- DAO: `@Dao interface`; reads return `Flow`, writes are `suspend`; `@Transaction` on queries returning relation classes

- Database: `@Database(entities, version, exportSchema)` on an abstract `RoomDatabase` with one abstract function per DAO
- Hilt: `@HiltAndroidApp` on the Application (registered in the manifest), `@AndroidEntryPoint` on the activity, `@Module @InstallIn(SingletonComponent)` with `@Provides @Singleton`
- Repository: `@Singleton class ... @Inject constructor`; the only thing screens talk to
- Seeding: `ensureSeeded()` inserts the four types when the table is empty
- Multi-step writes go inside `db.withTransaction { }`

NOT YET VERIFIED

Checked (one web search on 2026-11-19): that Room 2.x uses `room-runtime`, `room-compiler` through `ksp` and optionally `room-ktx`; that Room 3.0 exists with `androidx.room3` packages, requires KSP and has different callback signatures; that the (deprecated) Hilt codelab provides the database from a `SingletonComponent` module with `@Singleton`, and the DAO from a second `@Provides` function.

Not checked, from general knowledge: every artifact name outside Room (`hilt-android`, `hilt-android-compiler`, `hilt-navigation-compose`, the plugin ids); that Room 2.8 still ships `room-ktx` with `withTransaction`; the `NOCASE` comparison in `findByName`; `exportSchema` and `room.schemaLocation` behaviour; the Gradle layout for plugins; and all code, none of which was compiled. Version numbers are deliberately left as placeholders because I could not confirm a compatible Room, Hilt, KSP and Kotlin set.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Android Developers, Room release notes; Save data in a local database using Room; Migrate from Room 2.x to Room 3.0; (Deprecated) Using Hilt in your Android app. Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: Android (Kotlin & Compose) 4 builds the list screen: a view model that exposes the repository's `Flow` as `StateFlow`, a `LazyColumn` of items, type filter chips and an empty state.

Personal Catalogue: Android (Kotlin & Compose)

Course 1 · Chapter 4 · ViewModel, LazyColumn and Filter Chips

The database 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 is also where the course's screen pattern is set, and every later screen repeats it: **a ViewModel holds the state, a stateful composable collects it, and a stateless composable draws it.**

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Nothing was compiled or run** (no Android Studio here). My search confirmed the state-collection pattern in Google's documentation but **found nothing on Material 3's** `FilterChip`, so its parameters are from my memory and are the likeliest place for a compile error. Check the Material 3 component page if your IDE complains.

The Pattern: Three Parts per Screen

PART	ANDROID	WEB CATALOGUE EQUIVALENT
State holder	`ListViewModel` exposing a `StateFlow<ListUiState>`	The Django view (gathers the data), or a React hook
Stateful composable	`ListScreen`: gets the view model, collects the state	A container component
Stateless composable	`ListContent`: takes plain values and callbacks, draws them	The template, or a presentational component

The split pays off twice. The stateless composable can be previewed and tested with made-up data, and the view model survives screen rotation, so the selected filter does not reset when the phone turns.

The UI State and the ViewModel

Describe everything the screen needs in one immutable class. The view model combines three flows (the items, the types, and which type is selected) into it. Google's recommended way to turn that into a `StateFlow` is `stateIn` with `SharingStarted.WhileSubscribed(5000)`: the flow stays active for five seconds after the screen disappears, so a rotation does not restart the work.

```
package com.example.catalogue.ui.list

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.example.catalogue.data.CatalogueRepository
import com.example.catalogue.data.Item
import com.example.catalogue.data.ItemType
import com.example.catalogue.data.ItemWithDetails
import dagger.hilt.android.lifecycle.HiltViewModel
import javax.inject.Inject
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.combine
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch

data class ListUiState(
    val isLoading: Boolean = true,
    val types: List<ItemType> = emptyList(),
    val selectedTypeId: Long? = null,        // null means "All"
    val items: List<ItemWithDetails> = emptyList(),   // already filtered
    val totalCount: Int = 0                    // before filtering, to choose the empty m
)

@HiltViewModel
class ListViewModel @Inject constructor(
    private val repository: CatalogueRepository
) : ViewModel() {

    private val selectedTypeId = MutableStateFlow<Long?>(null)

    val uiState: StateFlow<ListUiState> = combine(
        repository.items,
        repository.itemTypes,
        selectedTypeId
    ) { items, types, selected ->
        ListUiState(
            isLoading = false,
```

```
                types = types,
                selectedTypeId = selected,
                items = if (selected == null) items else items.filter { it.item.itemTypeId ==
                totalCount = items.size
            )
        }.stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5_000),
            initialValue = ListUiState()
        )

        init {
            viewModelScope.launch { repository.ensureSeeded() }
        }

        fun onTypeSelected(typeId: Long?) {
            selectedTypeId.value = typeId
        }

        // TEMPORARY: lets you see rows before the form exists (Chapter 6 removes this).
        fun addSampleItem() {
            viewModelScope.launch {
                val type = uiState.value.types.firstOrNull() ?: return@launch
                repository.saveItem(
                    Item(title = "Sample item", creator = "Sample creator", itemTypeId = typ
                    tagIds = emptySet()
                )
            }
        }
    }
```

Two details. `ensureSeeded()` (Chapter 3) runs when the view model is created; the type chips appear as soon as the insert finishes, because `itemTypes` is a `Flow`. And the filtering is done in memory on the already-loaded list: for a personal collection of a few thousand items that is simple and fast. Chapter 8 moves search into the database, where it belongs once there are several conditions.

The Stateful Screen

The stateful composable gets the view model from Hilt (`hiltViewModel()`, from the `hilt-navigation-compose` dependency you added in Chapter 3) and turns the `StateFlow` into Compose state. `collectAsStateWithLifecycle()` is Google's recommended way to do that: it only collects while the screen is at least *started*, so a backgrounded app does no needless work. It needs the `androidx.lifecycle:lifecycle-runtime-compose` artifact; if your project does not have it, add it to the version catalog like the others.

```
package com.example.catalogue.ui.list

import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle

@Composable
fun ListScreen(
    onAddClick: () -> Unit,
    onItemClick: (Long) -> Unit,
    viewModel: ListViewModel = hiltViewModel()
) {
    val state by viewModel.uiState.collectAsStateWithLifecycle()

    ListContent(
        state = state,
        onTypeSelected = viewModel::onTypeSelected,
        onAddClick = viewModel::addSampleItem,   // TEMPORARY; Chapter 6 uses onAddClick
        onItemClick = onItemClick
    )
}
```

The Stateless Content

Now the drawing. `ListContent` receives plain values and lambdas, and knows nothing about view models. This is **state hoisting**: the state lives higher up, and the composable below it only displays state and reports events (Chapter 6 returns to this idea for the form). The pieces are a `Scaffold` with a top bar and a "+" button, a row of filter chips, and either a `LazyColumn` or an empty-state message.

```
package com.example.catalogue.ui.list

import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.LazyRow
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.clickable
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Add
import androidx.compose.material3.*
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import com.example.catalogue.data.ItemWithDetails

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ListContent(
    state: ListUiState,
    onTypeSelected: (Long?) -> Unit,
    onAddClick: () -> Unit,
    onItemClick: (Long) -> Unit
) {
    Scaffold(
        topBar = { TopAppBar(title = { Text("My Catalogue") }) },
        floatingActionButton = {
            FloatingActionButton(onClick = onAddClick) {
                Icon(Icons.Filled.Add, contentDescription = "Add item")
            }
        }
    ) { innerPadding ->
        Column(Modifier.padding(innerPadding)) {
            TypeFilterRow(state, onTypeSelected)

            when {
                state.isLoading -> Unit
                state.items.isEmpty() -> EmptyMessage(hasAnyItems = state.totalCount > 0
                else -> LazyColumn(
                    contentPadding = PaddingValues(16.dp),
                    verticalArrangement = Arrangement.spacedBy(8.dp)
                ) {
                    items(state.items, key = { it.item.id }) { row ->
```

```
                        ItemRow(row, onClick = { onItemClick(row.item.id) })
                    }
                }
            }
        }
    }
}

@Composable
private fun TypeFilterRow(state: ListUiState, onTypeSelected: (Long?) -> Unit) {
    LazyRow(
        contentPadding = PaddingValues(horizontal = 16.dp),
        horizontalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        item {
            FilterChip(
                selected = state.selectedTypeId == null,
                onClick = { onTypeSelected(null) },
                label = { Text("All") }
            )
        }
        items(state.types, key = { it.id }) { type ->
            FilterChip(
                selected = state.selectedTypeId == type.id,
                onClick = { onTypeSelected(type.id) },
                label = { Text(type.name) }
            )
        }
    }
}

@Composable
private fun ItemRow(row: ItemWithDetails, onClick: () -> Unit) {
    Card(modifier = Modifier.fillMaxWidth().clickable(onClick = onClick)) {
        Column(Modifier.padding(16.dp)) {
            Text(row.item.title, style = MaterialTheme.typography.titleMedium, maxLines =
            val byline = listOfNotNull(
                row.item.creator.ifBlank { null },
                row.item.releaseYear?.toString()
            ).joinToString(" · ")
            if (byline.isNotEmpty()) {
                Text(byline, style = MaterialTheme.typography.bodyMedium)
            }
            Text(
                text = (listOf(row.type.name) + row.tags.map { it.name }).joinToString("
```

```
                style = MaterialTheme.typography.labelMedium,
                color = MaterialTheme.colorScheme.primary
            )
        }
    }
}

@Composable
private fun EmptyMessage(hasAnyItems: Boolean) {
    Box(Modifier.fillMaxSize().padding(32.dp), contentAlignment = Alignment.Center) {
        Text(
            text = if (hasAnyItems) "Nothing of this type yet."
                   else "Your catalogue is empty. Tap + to add your first item.",
            style = MaterialTheme.typography.bodyLarge
        )
    }
}
```

Points worth knowing. `items(state.items, key = { it.item.id })` gives each row a **stable key**: Compose uses it to keep a row's identity when the list changes (an item deleted from the middle, say) instead of treating every following row as new. `LazyColumn` only composes the rows that are on screen, which is why it is the right choice for a list of unknown length, unlike a plain `Column` that would build every row up front. The two empty messages come from `totalCount`: "Nothing of this type yet" appears when a filter hides everything, "Your catalogue is empty" only when there really is nothing. The icon needs `Icons.Filled.Add`, which lives in the core icon set; the larger extended set is not needed for this app.

Showing It

Replace the body of `CatalogueApp` from Chapter 1 so that it displays the list. (Chapter 5 will replace this again with navigation.) Remove the unused imports the IDE greys out.

```
@Composable
fun CatalogueApp() {
    ListScreen(
        onAddClick = { /* Chapter 6 */ },
        onItemClick = { /* Chapter 5 */ }
```

```
        )
    }
```

Run it. You should see the top bar, four chips plus "All", and the empty message. Tap the "+" button: for now it inserts a sample item, which appears immediately with no refresh code, because the list is collecting a `Flow` that Room re-emits on every change. That is the whole payoff of the architecture so far.

> **COMPARE WITH THE IOS COURSE**
>
> In *Personal Catalogue: iOS (Swift & SwiftUI)* Chapter 4, `@Query` in the view replaces the view model and the flow plumbing, and a `List` with a segmented `Picker` replaces the `LazyColumn` and chips. The screens look alike; the Android version has more moving parts because it keeps the state holder and the drawing apart.

Challenges

EXERCISE 1

Add `ListViewModel`, `ListScreen` and `ListContent`, update `CatalogueApp`, and run it. Tap "+" three times, tap a type 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 top bar title to show the number of items currently listed, for example "My Catalogue (3)". Decide where the number should come from: the view model, or computed in `ListContent` from `state.items`.

View solution

EXERCISE 3

Rotate the emulator while a type chip is selected. Explain why the chip stays selected, and what would happen if `selectedTypeId` were stored with `remember { mutableStateOf(...) }` inside `ListContent`

instead. Then name the Compose function that would keep it across rotation without a view model.

View solution

CHAPTER 4 QUICK REFERENCE

- Screen pattern: ViewModel (state) → stateful composable (collects) → stateless composable (draws, takes values and lambdas)
- `combine(flowA, flowB, flowC) { ... }.stateIn(viewModelScope, SharingStarted.WhileSubscribed(5_000), initial)` builds the `StateFlow`
- `@HiltViewModel` + `@Inject constructor`; in a screen: `hiltViewModel()`
- `val state by vm.uiState.collectAsStateWithLifecycle()` (needs `lifecycle-runtime-compose`)
- `LazyColumn { items(list, key = { it.id }) { ... } }`: only visible rows are composed; keys keep row identity
- `FilterChip(selected, onClick, label)` in a `LazyRow` for the type filter; `null` type id means All
- Two empty states: no items at all, and no items of the chosen type (decide with `totalCount`)

NOT YET VERIFIED

Checked (one web search on 2026-11-19): that `collectAsStateWithLifecycle` is the recommended way to collect flows in Compose and collects from the STARTED state; that the documented pattern builds UI state with `stateIn` and `WhileSubscribed(5000)` (the five seconds covering configuration changes); that `hiltViewModel()` is paired with it in Google's state-holder guide; and that a documented `LazyColumn` example sets a `key`.
Not checked, from general knowledge: `FilterChip` (the search found nothing on it; parameters and any opt-in annotation requirement may differ in your Material 3 version); `TopAppBar` needing `@OptIn(ExperimentalMaterial3Api)` in some versions; `Icons.Filled.Add` being in the core icon set; the artifact name `lifecycle-runtime-compose`; the wildcard imports resolving correctly; the `combine` overload for three flows; and all code, none of which was compiled.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): collectAsStateWithLifecycle API reference; State holders and UI state; Where to hoist state; Filter a list while typing; Use Kotlin coroutines with lifecycle-aware components. Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: Android (Kotlin & Compose) 5 adds Navigation Compose: tapping a row opens a detail screen for that item, passing its id, with a delete button and a confirmation dialog.

Personal Catalogue: Android (Kotlin & Compose)

Course 1 · Chapter 5 · Navigation, Detail and 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 **Navigation Compose**, and lets you delete an item after a confirmation. You will also see how a screen is handed the *id* of what to display, rather than the whole item, which is the habit that keeps screens independent of each other.

> **WHAT WAS RUN, AND WHAT WAS NOT, AND A NAVIGATION WARNING**
>
> **Nothing was compiled or run** (no Android Studio here). My search confirmed the type-safe navigation API in Google's documentation, but **found nothing on Material 3's** `AlertDialog`, which is therefore written from memory. Also, the search results mention **Navigation 3**, a newer, separate navigation library. This course uses the established **Navigation Compose** (Navigation 2.x) library, as the outline promises, so if your tutorial or project template uses Navigation 3, the idea is the same but the code is not.

Navigation in One Picture

IDEA	NAVIGATION COMPOSE	WEB CATALOGUE EQUIVALENT
A screen's address	A **route**: a Kotlin class, e.g. `DetailRoute(itemId = 7)`	A URL, `/items/7/`
The route table	A `NavHost` with one `composable<Route>` per screen	`urls.py` or the router
Going to a page	`navController.navigate(DetailRoute(7))`	Following a link
The Back button	The back stack: `navController.popBackStack()`	The browser's back button
Reading the id	`toRoute<DetailRoute>()`	`request.GET` or the URL parameter

Older tutorials write routes as strings such as `"detail/{itemId}"`. Google's current guide recommends **type-safe routes** instead: each route is a `@Serializable` object or class, so a mistyped argument is a compile error rather than a crash at runtime.

Dependencies

Type-safe routes use Kotlin serialization. In the version catalog add the `navigation-compose` library, the `kotlinx-serialization-json` library and the `org.jetbrains.kotlin.plugin.serialization` plugin; apply the plugin and add both libraries in `app/build.gradle.kts`, exactly as you did for KSP and Hilt in Chapter 3. As before, **use the current versions from the release pages**; the serialization plugin version normally equals your Kotlin version. Sync Gradle afterwards.

The Routes

Put both routes in `ui/navigation/Routes.kt`. The list needs no data, so it is an `object`; the detail screen needs one argument, so it is a `data class`.

```
package com.example.catalogue.ui.navigation

import kotlinx.serialization.Serializable

@Serializable object ListRoute

@Serializable data class DetailRoute(val itemId: Long)
```

The NavHost

The `NavHost` is the route table. It owns a `NavController` (created with `rememberNavController()`), starts at the list, and maps each route to a screen. Callbacks such as `onItemClick` are where navigation actually happens: screens never see the controller, only plain lambdas, which keeps them previewable and testable.

```
package com.example.catalogue.ui.navigation

import androidx.compose.runtime.Composable
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.rememberNavController
import com.example.catalogue.ui.detail.DetailScreen
```

```
import com.example.catalogue.ui.list.ListScreen

@Composable
fun CatalogueNavHost() {
    val navController = rememberNavController()

    NavHost(navController = navController, startDestination = ListRoute) {
        composable<ListRoute> {
            ListScreen(
                onAddClick = { /* Chapter 6 */ },
                onItemClick = { id -> navController.navigate(DetailRoute(id)) }
            )
        }
        composable<DetailRoute> {
            DetailScreen(
                onBack = { navController.popBackStack() },
                onEditClick = { /* Chapter 6 */ }
            )
        }
    }
}
```

Update `MainActivity`'s `setContent` to call `CatalogueNavHost()` inside `CatalogueTheme`, and delete the now-unused `CatalogueApp` from Chapter 4. Note that `DetailScreen` is called with no id: the view model reads it for itself, as the next section shows.

The Detail ViewModel

A `NavBackStackEntry` carries the route object, and Hilt gives the view model a `SavedStateHandle` holding the same arguments. Per Google's documentation, the route can be rebuilt from that handle with `toRoute<DetailRoute>()`, so the id arrives type-checked. The view model exposes one of three states, which stops the screen having to guess between "still loading" and "that item does not exist".

```
package com.example.catalogue.ui.detail

import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.navigation.toRoute
```

```
import com.example.catalogue.data.CatalogueRepository
import com.example.catalogue.data.ItemWithDetails
import com.example.catalogue.ui.navigation.DetailRoute
import dagger.hilt.android.lifecycle.HiltViewModel
import javax.inject.Inject
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch

sealed interface DetailUiState {
    data object Loading : DetailUiState
    data object NotFound : DetailUiState
    data class Found(val details: ItemWithDetails) : DetailUiState
}

@HiltViewModel
class DetailViewModel @Inject constructor(
    savedStateHandle: SavedStateHandle,
    private val repository: CatalogueRepository
) : ViewModel() {

    private val itemId: Long = savedStateHandle.toRoute<DetailRoute>().itemId

    val uiState: StateFlow<DetailUiState> = repository.observeItem(itemId)
        .map { found -> if (found == null) DetailUiState.NotFound else DetailUiState.Fou
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5_000), DetailUiState.Lo

    fun delete(onDeleted: () -> Unit) {
        val current = (uiState.value as? DetailUiState.Found)?.details ?: return
        viewModelScope.launch {
            repository.deleteItem(current.item)
            onDeleted()
        }
    }
}
```

Two details. Deleting the item makes `observeItem` emit `null` at once, so the screen may briefly flip to `NotFound` before `onDeleted()` pops it off the back stack; that flash is harmless, and `NotFound` also covers the real case of a stale id. And thanks to the `CASCADE` rule in Chapter 2, deleting an item removes its tag links but leaves the tags themselves, with no extra code here.

The Detail Screen

Same pattern as Chapter 4: a stateful `DetailScreen` that collects the state, and a stateless `DetailContent` that draws it. The confirmation dialog is *UI-only state*: nothing but this screen cares whether it is open, so it lives in the composable with `rememberSaveable` (which, as Chapter 4's third exercise showed, survives rotation).

```
package com.example.catalogue.ui.detail

import androidx.compose.foundation.layout.*
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.verticalScroll
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material.icons.filled.Delete
import androidx.compose.material.icons.filled.Edit
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.runtime.saveable.rememberSaveable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import com.example.catalogue.data.ItemWithDetails
import java.text.DateFormat
import java.util.Date

@Composable
fun DetailScreen(
    onBack: () -> Unit,
    onEditClick: () -> Unit,
    viewModel: DetailViewModel = hiltViewModel()
) {
    val state by viewModel.uiState.collectAsStateWithLifecycle()
    DetailContent(
        state = state,
        onBack = onBack,
        onEditClick = onEditClick,
        onConfirmDelete = { viewModel.delete(onDeleted = onBack) }
    )
}
```

```kotlin
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun DetailContent(
    state: DetailUiState,
    onBack: () -> Unit,
    onEditClick: () -> Unit,
    onConfirmDelete: () -> Unit
) {
    var showConfirm by rememberSaveable { mutableStateOf(false) }
    val found = state is DetailUiState.Found

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text("Details") },
                navigationIcon = {
                    IconButton(onClick = onBack) {
                        Icon(Icons.AutoMirrored.Filled.ArrowBack, contentDescription = "
                    }
                },
                actions = {
                    if (found) {
                        IconButton(onClick = onEditClick) {
                            Icon(Icons.Filled.Edit, contentDescription = "Edit item")
                        }
                        IconButton(onClick = { showConfirm = true }) {
                            Icon(Icons.Filled.Delete, contentDescription = "Delete item"
                        }
                    }
                }
            )
        }
    ) { innerPadding ->
        Box(Modifier.fillMaxSize().padding(innerPadding)) {
            when (state) {
                DetailUiState.Loading -> Unit
                DetailUiState.NotFound -> Text(
                    "This item no longer exists.",
                    modifier = Modifier.align(Alignment.Center)
                )
                is DetailUiState.Found -> ItemDetails(state.details)
            }
        }
    }
```

```
    if (showConfirm && state is DetailUiState.Found) {
        AlertDialog(
            onDismissRequest = { showConfirm = false },
            title = { Text("Delete this item?") },
            text = { Text("\"${state.details.item.title}\" will be removed from your cat
            confirmButton = {
                TextButton(onClick = { showConfirm = false; onConfirmDelete() }) { Text(
            },
            dismissButton = {
                TextButton(onClick = { showConfirm = false }) { Text("Cancel") }
            }
        )
    }
}

@Composable
private fun ItemDetails(details: ItemWithDetails) {
    val item = details.item
    Column(
        modifier = Modifier.fillMaxSize().verticalScroll(rememberScrollState()).padding(
        verticalArrangement = Arrangement.spacedBy(16.dp)
    ) {
        Text(item.title, style = MaterialTheme.typography.headlineSmall)
        Field("Type", details.type.name)
        Field("Creator", item.creator)
        Field("Format", item.formatDetail)
        Field("Year", item.releaseYear?.toString().orEmpty())
        Field("Tags", details.tags.joinToString(", ") { it.name })
        Field("Notes", item.notes)
        Field("Added", DateFormat.getDateInstance().format(Date(item.createdAt)))
    }
}

/** One labelled value; shows nothing at all when the value is blank. */
@Composable
private fun Field(label: String, value: String) {
    if (value.isBlank()) return
    Column {
        Text(label, style = MaterialTheme.typography.labelMedium, color = MaterialTheme.
        Text(value, style = MaterialTheme.typography.bodyLarge)
    }
}
```

The dialog text includes the item's title, so the user sees *what* they are about to delete. The confirm button is a plain `TextButton`; a destructive action should never happen on a single accidental tap. `Field` returns early when a value is blank, so an item with no year or notes simply has no empty rows; in Chapter 6 these labels will change with the item type ("Author" for a book, "Artist" for a CD, "Director" for a film).

Try It

Run the app, add a sample item with "+", tap its row (the detail screen appears), press the system Back button or the arrow (back to the list), open it again and tap the bin icon, then Cancel, then the bin icon again and Delete. The row should vanish from the list with no refresh code, because the list is collecting a `Flow`.

COMPARE WITH THE IOS COURSE

In *Personal Catalogue: iOS (Swift & SwiftUI)* Chapter 5, a `NavigationStack` with `NavigationLink` replaces the `NavHost`, the value you link with plays the part of the route, and `.confirmationDialog` or `.alert` replaces `AlertDialog`. Both platforms pass an *identifier* or a lightweight value, not a copy of the whole record.

Challenges

EXERCISE 1

Add the dependencies, `Routes.kt`, `CatalogueNavHost`, the detail view model and screen, and update `MainActivity`. Run the app and perform the full add, open, back, delete-with-cancel, delete sequence. Write down which step you would expect to fail first if you forgot the serialization plugin.

View solution

EXERCISE 2

Rotate the emulator while the delete dialog is open. Does the dialog survive? Explain why, and say what would change if `showConfirm` used plain `remember`.

View solution

EXERCISE 3

The list passes only an `Long` id to the detail screen. Give two reasons that is better than passing the whole `ItemWithDetails`, and describe one bug you would get if the detail screen kept its own copy of the item after the user edited it elsewhere.

View solution

CHAPTER 5 QUICK REFERENCE

- Routes: `@Serializable object ListRoute`, `@Serializable data class DetailRoute(val itemId: Long)` (needs the serialization plugin and library)
- Route table: `NavHost(navController, startDestination = ListRoute) { composable<ListRoute> { ... } }`
- Go: `navController.navigate(DetailRoute(id))`; back: `navController.popBackStack()`
- Read the argument in a Hilt view model: `savedStateHandle.toRoute<DetailRoute>().itemId`
- Screens receive lambdas, never the `NavController`
- Three detail states: Loading, NotFound, Found
- Dialog visibility is UI-only state: `rememberSaveable { mutableStateOf(false) }`; `AlertDialog(onDismissRequest, title, text, confirmButton, dismissButton)`
- Delete cascades to the tag links (Chapter 2) but never deletes tags

NOT YET VERIFIED

Checked (one web search on 2026-11-19): that Google's guide recommends `@Serializable` routes with `composable<Route>`; that `toRoute()` converts a back-stack entry and `SavedStateHandle.toRoute<T>()` reads the route in a view model; that `hiltViewModel()` works with Compose navigation; and that a separate Navigation 3 library exists.
Not checked, from general knowledge: `AlertDialog` and its parameters (the search found nothing on it); that `navigate(DetailRoute(id))` and `startDestination = ListRoute` take route objects in your version; that a view model built by `hiltViewModel()` inside a `composable<DetailRoute>` receives the route arguments in its `SavedStateHandle` (the pattern is standard, but I did not confirm it for Hilt specifically); the icons `ArrowBack` (the `automirrored` package), `Delete` and `Edit` being in the core icon set; the `@OptIn` requirement; and all code, none of which was compiled.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Type safety in Kotlin DSL and Navigation Compose; Migrating to Type-Safe Navigation in Compose and Navigation 2; Navigation with Compose; SavedStateHandle API reference; Use Hilt with other Jetpack libraries; Migrate from Navigation 2 to Navigation 3. Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: Android (Kotlin & Compose) 6 builds the add and edit form: one screen whose labels change with the item type, input validation, and state hoisting for text fields. It replaces Chapter 4's temporary "+" behaviour and makes the Edit button work.

Personal Catalogue: Android (Kotlin & Compose)

Course 1 · Chapter 6 · Forms, Validation and Hoisted State

So far the only way to put data in is a temporary button that inserts "Sample item". This chapter replaces it with the real thing: **one form screen used 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 the state kept where it belongs. 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 was compiled or run** (no Android Studio here). My search confirmed how `OutlinedTextField` shows errors, how a number keyboard is requested, and Google's pattern of hoisting a text field's value and error flag into a view model. It did **not** give a complete example of a form, so the form's structure is my own design from those parts. The exposed-dropdown API was reported to be changing between Material 3 versions, so this chapter deliberately avoids it and picks the type with chips.

One Form, Two Jobs

Adding and editing are the same screen with different starting values. A web developer will recognise this: Django's `ModelForm` is used for both create and update views. Here a route argument decides which: an item id of `0` means "new item" (which is also what Room's auto-generated key defaults to), any other id means "edit that item".

CONCERN	DJANGO MODELFORM	THIS FORM
Create vs update	Two URLs, one form class	One route, `FormRoute(itemId)`, 0 = new
Starting values	`instance=item`	The view model loads the item into its state
Validation	`form.is_valid()`, errors in the template	Computed in the view model, shown with `isError` and `supportingText`
Per-type labels	Template logic or a form subclass	One small function, `labelsFor(type)`

CONCERN	DJANGO MODELFORM	THIS FORM
Saving	`form.save()`	`repository.saveItem(item, tagIds)` (Chapter 3)

The Route

Add one line to `Routes.kt`. A default value lets the list's "+" button write `FormRoute()` for a new item.

```
@Serializable data class FormRoute(val itemId: Long = 0L)
```

Labels That Follow the Type

The spec's generic `creator` and `formatDetail` fields get friendlier names per type. A small function keeps this in one place, so the form, and later the detail screen, share it. It matches on the seeded type names from Chapter 3, which is fine for a fixed list; if the list could grow, an enum or a column on `ItemType` would be the sturdier design.

```
package com.example.catalogue.ui.form

data class FieldLabels(val creator: String, val format: String)

fun labelsFor(typeName: String?): FieldLabels = when (typeName) {
    "Book" -> FieldLabels(creator = "Author", format = "Format (hardcover, paperback...)
    "CD" -> FieldLabels(creator = "Artist", format = "Format (album, single...)")
    "DVD", "Blu-ray" -> FieldLabels(creator = "Director", format = "Edition or region co
    else -> FieldLabels(creator = "Creator", format = "Format")
}
```

The Form State and ViewModel

State hoisting means the text a user is typing is not stored inside the text field's composable; the field is handed its current `value` and told, through `onValueChange`, when the user types. Google's validation guide does exactly this, keeping the value and its error flag in a view model.

That is why the form can be pre-filled for an edit, why it can be validated before saving, and why rotating the phone does not lose what was typed.

Two design choices. The year is held as **text** while it is being typed (an empty box is not a number) and only converted when saving. And errors are only shown after the first press of Save, so the form does not shout "Title is required" at someone who has not started typing.

```
package com.example.catalogue.ui.form

import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.navigation.toRoute
import com.example.catalogue.data.CatalogueRepository
import com.example.catalogue.data.Item
import com.example.catalogue.data.ItemType
import com.example.catalogue.ui.navigation.FormRoute
import dagger.hilt.android.lifecycle.HiltViewModel
import javax.inject.Inject
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.flow.update
import kotlinx.coroutines.launch
import java.util.Calendar

data class FormUiState(
    val isEdit: Boolean = false,
    val types: List<ItemType> = emptyList(),
    val typeId: Long? = null,
    val title: String = "",
    val creator: String = "",
    val formatDetail: String = "",
    val yearText: String = "",
    val notes: String = "",
    val showErrors: Boolean = false,
    val isSaving: Boolean = false
) {
    val typeName: String? get() = types.firstOrNull { it.id == typeId }?.name
    val titleError: Boolean get() = title.isBlank()
    val yearError: Boolean get() = yearText.isNotEmpty() && yearText.toIntOrNull() !in 1
    val isValid: Boolean get() = !titleError && !yearError && typeId != null
}
```

```kotlin
private fun currentYear() = Calendar.getInstance().get(Calendar.YEAR)

@HiltViewModel
class FormViewModel @Inject constructor(
    savedStateHandle: SavedStateHandle,
    private val repository: CatalogueRepository
) : ViewModel() {

    private val itemId: Long = savedStateHandle.toRoute<FormRoute>().itemId
    private val _state = MutableStateFlow(FormUiState(isEdit = itemId != 0L))
    val uiState: StateFlow<FormUiState> = _state.asStateFlow()

    // Kept outside the visible state: not edited here, but must survive a save (see bel
    private var originalCreatedAt: Long = System.currentTimeMillis()
    private var existingTagIds: Set<Long> = emptySet()

    init {
        viewModelScope.launch {
            repository.ensureSeeded()
            val types = repository.itemTypes.first { it.isNotEmpty() }
            _state.update { it.copy(types = types, typeId = it.typeId ?: types.first().i

            if (itemId != 0L) {
                repository.observeItem(itemId).first()?.let { found ->
                    originalCreatedAt = found.item.createdAt
                    existingTagIds = found.tags.map { it.id }.toSet()
                    _state.update {
                        it.copy(
                            typeId = found.item.itemTypeId,
                            title = found.item.title,
                            creator = found.item.creator,
                            formatDetail = found.item.formatDetail,
                            yearText = found.item.releaseYear?.toString().orEmpty(),
                            notes = found.item.notes
                        )
                    }
                }
            }
        }
    }

    fun onTypeChange(id: Long) = _state.update { it.copy(typeId = id) }
    fun onTitleChange(v: String) = _state.update { it.copy(title = v) }
    fun onCreatorChange(v: String) = _state.update { it.copy(creator = v) }
```

```
    fun onFormatChange(v: String) = _state.update { it.copy(formatDetail = v) }
    fun onNotesChange(v: String) = _state.update { it.copy(notes = v) }
    fun onYearChange(v: String) {
        if (v.length <= 4 && v.all(Char::isDigit)) _state.update { it.copy(yearText = v)
    }

    fun save(onSaved: () -> Unit) {
        val s = _state.value
        if (s.isSaving) return
        if (!s.isValid) {
            _state.update { it.copy(showErrors = true) }
            return
        }
        _state.update { it.copy(isSaving = true) }
        viewModelScope.launch {
            repository.saveItem(
                Item(
                    id = itemId,
                    title = s.title.trim(),
                    creator = s.creator.trim(),
                    formatDetail = s.formatDetail.trim(),
                    releaseYear = s.yearText.toIntOrNull(),
                    notes = s.notes.trim(),
                    createdAt = originalCreatedAt,
                    itemTypeId = s.typeId!!
                ),
                tagIds = existingTagIds
            )
            onSaved()
        }
    }
}
```

Read `save` carefully, because it contains the chapter's two quiet traps. First, `originalCreatedAt` is carried through, so editing an item does not make it look newly added. Second, `existingTagIds` is passed to `saveItem`: that function (Chapter 3) *replaces* the item's tag links, so an edit that passed an empty set would silently strip every tag. This chapter's form has no tag controls yet, so it must preserve them; Chapter 7 adds the controls and replaces the preserved set with the user's choice. The `isSaving` guard stops a double-tap from saving twice.

The Form Screen

```
package com.example.catalogue.ui.form

import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyRow
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.foundation.verticalScroll
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material3.*
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.input.KeyboardCapitalization
import androidx.compose.ui.text.input.KeyboardType
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle

@Composable
fun FormScreen(onDone: () -> Unit, viewModel: FormViewModel = hiltViewModel()) {
    val state by viewModel.uiState.collectAsStateWithLifecycle()
    FormContent(
        state = state,
        onBack = onDone,
        onSave = { viewModel.save(onSaved = onDone) },
        onTypeChange = viewModel::onTypeChange,
        onTitleChange = viewModel::onTitleChange,
        onCreatorChange = viewModel::onCreatorChange,
        onFormatChange = viewModel::onFormatChange,
        onYearChange = viewModel::onYearChange,
        onNotesChange = viewModel::onNotesChange
    )
}

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun FormContent(
    state: FormUiState,
    onBack: () -> Unit,
    onSave: () -> Unit,
```

```
    onTypeChange: (Long) -> Unit,
    onTitleChange: (String) -> Unit,
    onCreatorChange: (String) -> Unit,
    onFormatChange: (String) -> Unit,
    onYearChange: (String) -> Unit,
    onNotesChange: (String) -> Unit
) {
    val labels = labelsFor(state.typeName)

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text(if (state.isEdit) "Edit item" else "Add item") },
                navigationIcon = {
                    IconButton(onClick = onBack) {
                        Icon(Icons.AutoMirrored.Filled.ArrowBack, contentDescription = "
                    }
                },
                actions = { TextButton(onClick = onSave, enabled = !state.isSaving) { Te
            )
        }
    ) { innerPadding ->
        Column(
            modifier = Modifier
                .padding(innerPadding)
                .verticalScroll(rememberScrollState())
                .padding(16.dp),
            verticalArrangement = Arrangement.spacedBy(12.dp)
        ) {
            Text("Type", style = MaterialTheme.typography.labelLarge)
            LazyRow(horizontalArrangement = Arrangement.spacedBy(8.dp)) {
                items(state.types, key = { it.id }) { type ->
                    FilterChip(
                        selected = state.typeId == type.id,
                        onClick = { onTypeChange(type.id) },
                        label = { Text(type.name) }
                    )
                }
            }

            OutlinedTextField(
                value = state.title,
                onValueChange = onTitleChange,
                label = { Text("Title (required)") },
                singleLine = true,
```

```
                isError = state.showErrors && state.titleError,
                supportingText = { if (state.showErrors && state.titleError) Text("Pleas
                keyboardOptions = KeyboardOptions(capitalization = KeyboardCapitalizatio
                modifier = Modifier.fillMaxWidth()
            )
            OutlinedTextField(
                value = state.creator,
                onValueChange = onCreatorChange,
                label = { Text(labels.creator) },
                singleLine = true,
                keyboardOptions = KeyboardOptions(capitalization = KeyboardCapitalizatio
                modifier = Modifier.fillMaxWidth()
            )
            OutlinedTextField(
                value = state.formatDetail,
                onValueChange = onFormatChange,
                label = { Text(labels.format) },
                singleLine = true,
                modifier = Modifier.fillMaxWidth()
            )
            OutlinedTextField(
                value = state.yearText,
                onValueChange = onYearChange,
                label = { Text("Year (optional)") },
                singleLine = true,
                isError = state.showErrors && state.yearError,
                supportingText = { if (state.showErrors && state.yearError) Text("Enter
                keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
                modifier = Modifier.fillMaxWidth()
            )
            OutlinedTextField(
                value = state.notes,
                onValueChange = onNotesChange,
                label = { Text("Notes") },
                minLines = 3,
                modifier = Modifier.fillMaxWidth()
            )
        }
    }
}
```

Look at what the stateless `FormContent` does *not* have: no `remember`, no view model, no repository. Every character the user types goes through `onValueChange` to the view model and

returns as `state.title`. That is the full meaning of hoisted state, and it is why the labels update the instant a different type chip is tapped: the chip changes `typeId` in the state, `labelsFor` is called again on the next draw, and the field labels change while the typed text stays.

Wiring and Clean-up

Three small edits finish the chapter.

1. **NavHost.** Add the form destination, and make both buttons navigate to it:

```
composable<ListRoute> {
    ListScreen(
        onAddClick = { navController.navigate(FormRoute()) },
        onItemClick = { id -> navController.navigate(DetailRoute(id)) }
    )
}
composable<DetailRoute> {
    DetailScreen(
        onBack = { navController.popBackStack() },
        onEditClick = { id -> navController.navigate(FormRoute(id)) }
    )
}
composable<FormRoute> {
    FormScreen(onDone = { navController.popBackStack() })
}
```

2. **DetailScreen.** Its `onEditClick` must now carry the id. Change the parameter to `onEditClick: (Long) -> Unit` and, in `DetailScreen` (not `DetailContent`), pass `onEditClick = { (state as? DetailUiState.Found)?.let { onEditClick(it.details.item.id) } }` to `DetailContent`, which keeps its `() -> Unit`.
3. **List.** In `ListScreen` change `onAddClick = viewModel::addSampleItem` to `onAddClick = onAddClick`, then **delete** `addSampleItem` from `ListViewModel`, along with the now-unused `Item` import.

Run it: tap "+", pick CD (the label becomes "Artist"), fill in a title, and Save. Open the item, tap the pencil, change the title, Save, and the detail screen updates itself through its `Flow`. Try Save with an empty title to see the error message.

LIMITS OF THIS FORM

The typed text is held in the view model, which survives rotation but **not** the system killing the app in the background. Google's `SavedStateHandle` can cover that case, and a real app with long forms might use it; for a form this short it is a reasonable thing to leave out. The form also has no tag controls (Chapter 7) and no duplicate warning (Chapter 9).

COMPARE WITH THE IOS COURSE

In *Personal Catalogue: iOS (Swift & SwiftUI)* Chapter 6, a `Form` with `TextField`s bound to `@State` (or an `@Observable` draft object) plays the same role, and `Binding` is the iOS spelling of "value in, change out". The two courses use the same per-type labels and the same rules: title required, year optional but plausible.

Challenges

EXERCISE 1

Add the route, `labelsFor`, the form state, view model and screen, make the three wiring edits, and remove `addSampleItem`. Add one item of each type, and for each write down the label shown for the creator field. Then edit one of them and confirm its creation date on the detail screen did not change.

View solution

EXERCISE 2

Make the title field trim leading spaces as the user types, and make the year error appear *immediately* if the user types something outside 1000 to the coming year, rather than waiting for Save. Say which function you changed in each case and whether the change belongs in the view model or the composable.

View solution

EXERCISE 3

Explain what would go wrong in an edit if `save` passed `emptySet()` as the tag ids, and what would go wrong if it passed `System.currentTimeMillis()` as `createdAt`. For each, say which screen would show the damage and when.

View solution

CHAPTER 6 QUICK REFERENCE

- One form for add and edit: `@Serializable data class FormRoute(val itemId: Long = 0L)`; 0 means new
- Hoisted state: the field gets `value` and `onValueChange`; the view model owns the text; derived flags (`titleError`, `isValid`) are computed from it
- `OutlinedTextField(isError = ..., supportingText = { ... }, keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number))`
- Show errors only after the first Save attempt (`showErrors`)
- Keep the year as text while typing; convert with `toIntOrNull()` on save
- Labels per type from one function, `labelsFor(typeName)`
- Edits must preserve what the form does not show: `createdAt` and the existing tag ids
- `_state.update { it.copy(...) }` is the safe way to change a `MutableStateFlow`

NOT YET VERIFIED

Checked (one web search on 2026-11-19): that `isError` turns the field's label, indicator and icon to the error colour and tells accessibility services; that `supportingText` is a composable slot under the field; that `KeyboardOptions(keyboardType = KeyboardType.Number)` is the documented way to ask for a number keyboard; that Google's validation guide keeps the value and error flag in a view model and passes them to a stateless composable; and that `ExposedDropdownMenu` changed shape in a recent Material 3 alpha.

Not checked, from general knowledge: `KeyboardCapitalization.Words`; `minLines` on `OutlinedTextField`; that a `@Serializable` route class with a default value can be created as `FormRoute()` and read back with `toRoute` (the default is my assumption); the `first { it.isNotEmpty() }` startup logic and its behaviour if seeding is slow; the `MutableStateFlow.update` extension; and all code, none of which was compiled. I did not check what `OutlinedTextField` does with an `isError` field when `supportingText` is empty, so the layout may jump by a line when an error appears.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): OutlinedTextField API reference; KeyboardOptions API reference; Validate input as the user types; ExposedDropdownMenuBox API reference; ViewModel and State in Compose.

Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: Android (Kotlin & Compose) 7 adds tags: a managed list that never allows "Web Development" and "web development" to coexist, an autocomplete input on the form, tag chips, and the many-to-many writes that replace the preserved tag ids used in this chapter.

Personal Catalogue: Android (Kotlin & Compose)

Course 1 · Chapter 7 · Managed Tags and Many-to-Many Writes

Tags are what make a catalogue searchable by *meaning*: "sci-fi", "birthday present", "lent out". Chapter 2 gave them a table and a case-insensitive unique name, and Chapter 6's form carries an item's existing tags through an edit without letting you change them. This chapter adds the tag controls to the form: type a tag and pick from **suggestions of tags you already have**, see the chosen tags as removable **chips**, and save everything in one go. The aim is the spec's rule: a **managed** list, so "Web Development" and "web development" can never become two tags.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Nothing was compiled or run** (no Android Studio here). My search covered chips, `FlowRow` and the keyboard's Done action separately, and found no example combining them, so the tag editor is my own assembly. The sources **disagree about which opt-in annotation `FlowRow` needs** (`ExperimentalLayoutApi` in the API reference, an earlier alpha said `ExperimentalFoundationApi`), so if the IDE objects to the one used here, swap it.

What "Managed" Means

PROBLEM	UNMANAGED (FREE TEXT)	MANAGED (THIS CHAPTER)
Case variants	"Sci-Fi", "sci-fi", "SCI-FI" are three tags	One tag; the stored spelling is reused
Stray spaces	"sci-fi " and "sci-fi" differ	Names are trimmed and inner runs of spaces collapsed
Typos you've already fixed once	Typed again from scratch each time	Suggestions from existing tags as you type
Cancelled form	Stray empty tags left behind	Tags are only created when the item is saved

Three layers cooperate, and each catches what the one before misses: the **suggestions** make the right choice easy, the **normalising** function cleans what the user types, and the **database's case-insensitive unique index** (Chapter 2) is the final guarantee. Do not rely on the first two alone: a bug in them, or the JSON import in Chapter 10, would otherwise break the rule.

Step 1: Normalise Names and Fix `getOrCreateTag`

Chapter 3 ended with a warning: `getOrCreateTag` returned a tag with the wrong id if the insert was ignored. Fix that now, and add a function that cleans a name. Put the function in `data/TagNames.kt`:

```
package com.example.catalogue.data

private val whitespace = Regex("\\s+")

/** Trims the name and turns any run of whitespace into a single space. */
fun normaliseTagName(raw: String): String = raw.trim().replace(whitespace, " ")
```

In `CatalogueRepository` replace `getOrCreateTag` and add a second save function. The new one resolves tag *names* into ids and writes the item in **one transaction**, so a failure part-way leaves no half-saved item and no stray tags. To avoid nesting one `withTransaction` inside another, both public functions share a private helper that does the writes and leaves the transaction to the caller.

```
/** Returns the existing tag with this name (any case), or creates it. */
suspend fun getOrCreateTag(rawName: String): Tag {
    val name = normaliseTagName(rawName)
    tagDao.findByName(name)?.let { return it }
    tagDao.insert(Tag(name = name))              // -1 if another insert won the race; e
    return tagDao.findByName(name)!!
}

// Replaces the old saveItem so both versions share the same writes.
private suspend fun writeItem(item: Item, tagIds: Set<Long>): Long {
    val id = if (item.id == 0L) {
        itemDao.insert(item)
    } else {
        itemDao.update(item)
        item.id
    }
    itemTagDao.deleteForItem(id)
    itemTagDao.insertAll(tagIds.map { ItemTagCrossRef(itemId = id, tagId = it) })
    return id
}

suspend fun saveItem(item: Item, tagIds: Set<Long>): Long =
```

```
        db.withTransaction { writeItem(item, tagIds) }

    /** Saves the item and links it to the named tags, creating any that do not exist yet. */
    suspend fun saveItemWithTagNames(item: Item, tagNames: List<String>): Long =
        db.withTransaction {
            val ids = tagNames.filter { normaliseTagName(it).isNotEmpty() }
                .map { getOrCreateTag(it).id }
                .toSet()
            writeItem(item, ids)
        }
```

The `!!` in `getOrCreateTag` is justified because, after an insert that either succeeded or was ignored because the name already exists, a row with that name must be present; if it is not, something is badly wrong and a crash is more honest than a silent wrong id. Note that the unique index is case-insensitive, so `findByName("SCI-FI")` finds the stored "sci-fi" (Chapter 3 explained why).

Step 2: Tag State in the Form

The form keeps the chosen tags as a list of *names*, not ids. A brand-new tag has no id yet (and should not get one if the user cancels), so names are the one thing every tag, existing or new, has. Four small additions to `FormUiState`:

```
    // Add these properties to FormUiState:
    val allTags: List<Tag> = emptyList(),
    val tagNames: List<String> = emptyList(),   // the tags chosen for this item
    val tagQuery: String = "",                  // what is typed in the tag box

    // ...and this computed property inside the class body:
    val suggestions: List<String> get() {
        val q = normaliseTagName(tagQuery)
        if (q.isEmpty()) return emptyList()
        return allTags.map { it.name }
            .filter { name -> name.contains(q, ignoreCase = true) && tagNames.none { it.equa
            .take(5)
    }
```

(Import `com.example.catalogue.data.Tag` and `normaliseTagName`.) Suggestions are everything existing whose name *contains* what is typed, not already chosen, capped at five so the screen does not fill up. An exact "starts with" match is a common alternative; "contains" is more forgiving of the user's memory.

Step 3: Tag Logic in the ViewModel

Four changes to `FormViewModel`: load the list of all tags, load an edited item's tag names instead of its ids, handle adding and removing, and save through the new repository function.

```
// 1. Delete: private var existingTagIds ...   (no longer needed)

// 2. In init { }, before the existing viewModelScope.launch, keep allTags up to date:
viewModelScope.launch {
    repository.tags.collect { tags -> _state.update { it.copy(allTags = tags) } }
}

//    ...and in the edit branch, replace the existingTagIds line with a tagNames entry:
_state.update {
    it.copy(
        typeId = found.item.itemTypeId,
        title = found.item.title,
        creator = found.item.creator,
        formatDetail = found.item.formatDetail,
        yearText = found.item.releaseYear?.toString().orEmpty(),
        notes = found.item.notes,
        tagNames = found.tags.map { tag -> tag.name }.sorted()
    )
}

// 3. New functions:
fun onTagQueryChange(v: String) = _state.update { it.copy(tagQuery = v) }

fun addTag(raw: String) {
    val name = normaliseTagName(raw)
    if (name.isEmpty()) return
    _state.update { s ->
        // Reuse the stored spelling if the tag already exists in another case.
        val canonical = s.allTags.firstOrNull { it.name.equals(name, ignoreCase = true) }
        if (s.tagNames.any { it.equals(canonical, ignoreCase = true) }) {
            s.copy(tagQuery = "")
        } else {
```

```
            s.copy(tagNames = s.tagNames + canonical, tagQuery = "")
        }
    }
}

fun removeTag(name: String) = _state.update { s -> s.copy(tagNames = s.tagNames - name) }

// 4. In save(), replace the repository.saveItem(...) call:
repository.saveItemWithTagNames(
    Item(
        id = itemId,
        title = s.title.trim(),
        creator = s.creator.trim(),
        formatDetail = s.formatDetail.trim(),
        releaseYear = s.yearText.toIntOrNull(),
        notes = s.notes.trim(),
        createdAt = originalCreatedAt,
        itemTypeId = s.typeId!!
    ),
    tagNames = s.tagNames
)
```

The key line is `canonical`. If the user types "SCI-FI" and "sci-fi" already exists, the chip shows "sci-fi", the stored spelling, so the screen never displays a name that will not be saved. One small decision: any text left in the tag box at the moment of Save is *not* added; the Add button, the keyboard's Done key or a tapped suggestion adds a tag, and Exercise 3 asks you to consider changing that.

Step 4: The Tag Editor

One composable, in `ui/form/TagEditor.kt`, holds the whole control. Like everything in this course it is stateless: it is given the chosen tags, the query and the suggestions, and reports what the user does.

```
package com.example.catalogue.ui.form

import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.ExperimentalLayoutApi
import androidx.compose.foundation.layout.FlowRow
```

```kotlin
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.text.KeyboardActions
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Close
import androidx.compose.material3.*
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.input.ImeAction
import androidx.compose.ui.unit.dp

@OptIn(ExperimentalLayoutApi::class)
@Composable
fun TagEditor(
    selected: List<String>,
    query: String,
    suggestions: List<String>,
    onQueryChange: (String) -> Unit,
    onAdd: (String) -> Unit,
    onRemove: (String) -> Unit
) {
    Column(verticalArrangement = Arrangement.spacedBy(8.dp)) {
        OutlinedTextField(
            value = query,
            onValueChange = onQueryChange,
            label = { Text("Add a tag") },
            singleLine = true,
            keyboardOptions = KeyboardOptions(imeAction = ImeAction.Done),
            keyboardActions = KeyboardActions(onDone = { onAdd(query) }),
            trailingIcon = { TextButton(onClick = { onAdd(query) }, enabled = query.isNo
            modifier = Modifier.fillMaxWidth()
        )

        if (suggestions.isNotEmpty()) {
            Text("Existing tags", style = MaterialTheme.typography.labelMedium)
            FlowRow(horizontalArrangement = Arrangement.spacedBy(8.dp)) {
                suggestions.forEach { name ->
                    SuggestionChip(onClick = { onAdd(name) }, label = { Text(name) })
                }
            }
        }

        if (selected.isNotEmpty()) {
            FlowRow(horizontalArrangement = Arrangement.spacedBy(8.dp)) {
                selected.forEach { name ->
```

```
                    InputChip(
                        selected = true,
                        onClick = { onRemove(name) },
                        label = { Text(name) },
                        trailingIcon = { Icon(Icons.Filled.Close, contentDescription = "
                    )
                }
            }
        }
    }
}
```

`FlowRow` lays chips out left to right and wraps onto new lines, which is what a variable-length set of tags needs (a `LazyRow` would scroll sideways and hide most of them). The `ImeAction.Done` option puts a Done key on the keyboard and `onDone` runs when it is pressed. Each chosen tag's close icon has a `contentDescription` naming the tag, so a screen reader says "Remove tag sci-fi" rather than just "Close".

Step 5: Put It in the Form

Add three parameters to `FormContent` (`onTagQueryChange`, `onAddTag`, `onRemoveTag`), pass them from `FormScreen` as `viewModel::onTagQueryChange`, `viewModel::addTag` and `viewModel::removeTag`, and insert this block in the column between the Year field and the Notes field:

```
TagEditor(
    selected = state.tagNames,
    query = state.tagQuery,
    suggestions = state.suggestions,
    onQueryChange = onTagQueryChange,
    onAdd = onAddTag,
    onRemove = onRemoveTag
)
```

Run it. Add an item with tags "sci-fi" and "classic". Add a second item and type "sci" in the tag box: "sci-fi" should appear as a suggestion. Type "SCI-FI" in full and press Done: the chip should

read "sci-fi". Open the first item for editing: its two tags are there, and removing one and saving changes the item's tags without disturbing its creation date.

COMPARE WITH THE IOS COURSE

In *Personal Catalogue: iOS (Swift & SwiftUI)* Chapter 7, tags are a `@Model` related to items, and SwiftData maintains the join for you. The tag editor is the same idea in SwiftUI: a text field, a filtered list of suggestions and a wrapping row of chips. The uniqueness rule needs the same defence in depth: normalise the name, reuse an existing tag, and check again at save time.

Challenges

EXERCISE 1

Make all five changes and run the app. Test and record the result of each case: (a) add "sci-fi" to one item; (b) on another item type "SCI-FI" and press Done; (c) type " retro games " with extra spaces; (d) add the same tag twice to one item; (e) start adding a new tag, then press Cancel and check no new tag appears in the suggestions of the next form.

View solution

EXERCISE 2

Tags that no item uses are clutter. Write a DAO query that deletes them, a repository function that calls it, and decide *where* in the app it should be called. Explain why calling it inside the same transaction as `saveItemWithTagNames` is both convenient and slightly risky.

View solution

EXERCISE 3

A user types a tag, forgets to press Add, and presses Save: the tag is lost. Decide how `save()` should treat leftover text in `tagQuery`, implement it, and say whether your solution belongs in the view model or the composable, and why.

View solution

CHAPTER 7 QUICK REFERENCE

- Managed tags: suggestions (ease), `normaliseTagName` (cleaning), unique `NOCASE` index (guarantee)
- `getOrCreateTag`: find by name, else insert, then *re-read*; never trust the id from an ignored insert
- Form keeps chosen tags as *names*; tags are created only on Save, inside `db.withTransaction`
- `saveItemWithTagNames(item, names)` resolves names to ids and calls a private `writeItem` shared with `saveItem` (no nested transactions)
- Reuse the stored spelling: a typed "SCI-FI" becomes the existing "sci-fi"
- `FlowRow` wraps chips; `InputChip` shows a removable chip; `SuggestionChip` offers an existing tag
- `KeyboardOptions(imeAction = ImeAction.Done)` + `KeyboardActions(onDone = { ... })` for the Done key

NOT YET VERIFIED

Checked (one web search on 2026-11-19): that `InputChip` exists in Material 3 with a documented wrapping example using `FlowRow`; that `FlowRow` is marked experimental (with the sources disagreeing on the annotation); that `KeyboardActions.onDone` runs when the Done action is triggered; and that keyboard actions attach to text fields, not chips.

Not checked, from general knowledge: the exact parameters of `InputChip` and `SuggestionChip` in your Material 3 version; `ImeAction.Done` import paths; that `Icons.Filled.Close` is in the core icon set; a `TextButton` as `trailingIcon`; that Room's `withTransaction` calls from inside the transaction block (`getOrCreateTag` uses DAO calls, not a nested `withTransaction`) behave as expected; that the `NOCASE` collation applies in `findByName`; and all code, none of which was compiled.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): InputChip API reference; FlowRow API reference; KeyboardActions API reference; Configure text fields; Compose Foundation release notes. Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: Android (Kotlin & Compose) 8 adds search by title or creator, filtering by tag on the list screen, and the "do I already own this?" flow: a quick-check search designed for use in a shop.

Personal Catalogue: Android (Kotlin & Compose)

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

This is the chapter that delivers the app's reason for existing. You are standing in a shop holding a paperback; you type three letters of its title; the phone tells you *"You already have 1 match"* or *"Not in your catalogue"*, and in the second case one tap starts adding it with the title already filled in. Along the way you will move filtering from memory into the database, add a tag filter, and meet two Kotlin flow operators, `debounce` and `flatMapLatest`, that make live search behave.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Nothing was compiled or run** (no Android Studio here). Two searches were made for this chapter. The Kotlin side (`debounce`, `flatMapLatest`) is confirmed only by the official `Flow` reference and by forum discussions, which I treated as illustration rather than authority. The SQLite side (`LIKE` and `ESCAPE`) is confirmed from SQLite's own documentation. **Room's handling of nullable query parameters and of the SQL inside `@Query` was not found in any source**, so the DAO query is the likeliest place for a build error; Room checks the SQL when you build, so the message will name the problem.

Where the Filtering Happens

Chapter 4 filtered in memory: load every item, then discard the ones that do not match the selected type. That is fine for one condition, but search plus type plus tag would mean ever more Kotlin code that re-implements what a database does well. Now the filtering moves into one SQL query, and the view model just passes the user's choices to it.

CHOICE	IN MEMORY (CHAPTER 4)	IN THE DATABASE (THIS CHAPTER)
Cost as the collection grows	Reads every item on every change	Returns only the matches
Adding another filter	More `filter { }` lines	One more `AND` clause
Where the rules live	View model	DAO query (testable in Chapter 10)

CHOICE	IN MEMORY (CHAPTER 4)	IN THE DATABASE (THIS CHAPTER)
Web catalogue equivalent	Filtering a fetched array in JavaScript	`Item.objects.filter(Q(title__icontains=q) | Q(creator__icontains=q))`

Step 1: The Search Query

One DAO method takes three optional conditions: a text pattern, a type id and a tag id. A condition that is empty (`''` or `NULL`) is switched off by the `:param IS NULL OR ...` idiom, so a single query serves every combination. Add it to `ItemDao`:

```
@Transaction
@Query(
    """
    SELECT * FROM item
    WHERE (:typeId IS NULL OR itemTypeId = :typeId)
      AND (:pattern = '' OR title LIKE :pattern ESCAPE '\' OR creator LIKE :pattern ESCA
      AND (:tagId IS NULL OR id IN (SELECT itemId FROM item_tag WHERE tagId = :tagId))
    ORDER BY createdAt DESC
    """
)
fun search(pattern: String, typeId: Long?, tagId: Long?): Flow<List<ItemWithDetails>>

@Query("SELECT COUNT(*) FROM item")
fun observeCount(): Flow<Int>
```

Read the three conditions as a sentence: *the type matches (or no type was chosen), and the title or creator matches the pattern (or no pattern was typed), and the item carries the tag (or no tag was chosen).* The tag test uses a subquery on the link table from Chapter 2, which is why that table has its own index on `tagId`. Each `LIKE` ends with `ESCAPE '\'`, explained next. The count query lets the screen tell "your catalogue is empty" from "nothing matches" without loading everything.

The pattern, and why `ESCAPE` matters

SQLite's documentation says `LIKE` treats `%` as "any run of characters" and `_` as "any one character", and that the `ESCAPE` clause names a single character that, placed before `%`, `_` or

itself, makes it literal. Without that, someone searching for the title "100% Pure" would have the `%` read as a wildcard. SQLite also documents that `LIKE` ignores case **only for ASCII letters**: "dune" matches "Dune", but "é" does not match "É". For a catalogue with a few accented titles that is a real, small limit, and Exercise 2's notes mention a fix. Put the pattern builder in `data/Search.kt`:

```
package com.example.catalogue.data

/** Builds a LIKE pattern for "contains this text", or "" for no search at all. */
fun likePattern(raw: String): String {
    val q = raw.trim()
    if (q.isEmpty()) return ""
    val escaped = q
        .replace("\\", "\\\\")   // the escape character itself first
        .replace("%", "\\%")
        .replace("_", "\\_")
    return "%$escaped%"
}
```

Order matters: the escape character must be escaped *first*, or the backslashes added for `%` and `_` would themselves be doubled. In the repository add:

```
val itemCount: Flow<Int> = itemDao.observeCount()

fun search(query: String, typeId: Long?, tagId: Long?): Flow<List<ItemWithDetails>> =
    itemDao.search(likePattern(query), typeId, tagId)
```

Step 2: A Search That Waits for You to Stop Typing

A query on every keystroke is wasteful and makes the list flicker, so wait until the user pauses. `debounce(250)` does that: per the `Flow` reference, it drops a value if a newer one arrives within the timeout, and the latest value is always emitted. The second operator is `flatMapLatest`: it turns each set of filters into a database `Flow` and, when the filters change, *cancels* the old one and switches to the new. Without it, a slow earlier query could finish after a faster later one and overwrite the newer results.

Two details. The text box must show what is typed *immediately*, so the view model keeps the raw text in one flow and debounces a separate copy for searching. And chip taps should not be

delayed, so only the text is debounced.

Step 3: State and ViewModel

`ListUiState` gains the query, the tag list and the selected tag; `ListViewModel` is rewritten around three input flows. This replaces the version from Chapter 4 (and its in-memory filtering) completely.

```
package com.example.catalogue.ui.list

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.example.catalogue.data.CatalogueRepository
import com.example.catalogue.data.ItemType
import com.example.catalogue.data.ItemWithDetails
import com.example.catalogue.data.Tag
import dagger.hilt.android.lifecycle.HiltViewModel
import javax.inject.Inject
import kotlinx.coroutines.ExperimentalCoroutinesApi
import kotlinx.coroutines.FlowPreview
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.launch

data class ListUiState(
    val isLoading: Boolean = true,
    val types: List<ItemType> = emptyList(),
    val tags: List<Tag> = emptyList(),
    val query: String = "",
    val selectedTypeId: Long? = null,
    val selectedTagId: Long? = null,
    val items: List<ItemWithDetails> = emptyList(),
    val totalCount: Int = 0
) {
    val hasChipFilter: Boolean get() = selectedTypeId != null || selectedTagId != null
    val hasQuery: Boolean get() = query.isNotBlank()
}

private data class Controls(val query: String, val typeId: Long?, val tagId: Long?)

@OptIn(FlowPreview::class, ExperimentalCoroutinesApi::class)
@HiltViewModel
class ListViewModel @Inject constructor(
```

```
    private val repository: CatalogueRepository
) : ViewModel() {

    private val query = MutableStateFlow("")
    private val typeId = MutableStateFlow<Long?>(null)
    private val tagId = MutableStateFlow<Long?>(null)

    // What the controls show: immediate.
    private val controls = combine(query, typeId, tagId, ::Controls)

    // What the database is asked: the text waits for a pause, chips do not.
    private val results = combine(query.debounce(250), typeId, tagId, ::Controls)
        .distinctUntilChanged()
        .flatMapLatest { c -> repository.search(c.query, c.typeId, c.tagId) }

    val uiState: StateFlow<ListUiState> = combine(
        results, repository.itemTypes, repository.tags, repository.itemCount, controls
    ) { items, types, tags, count, c ->
        ListUiState(
            isLoading = false,
            types = types,
            tags = tags,
            query = c.query,
            selectedTypeId = c.typeId,
            selectedTagId = c.tagId,
            items = items,
            totalCount = count
        )
    }.stateIn(viewModelScope, SharingStarted.WhileSubscribed(5_000), ListUiState())

    init {
        viewModelScope.launch { repository.ensureSeeded() }
    }

    fun onQueryChange(v: String) { query.value = v }
    fun onTypeSelected(id: Long?) { typeId.value = id }
    fun onTagSelected(id: Long) { tagId.update { current -> if (current == id) null else
    fun clearFilters() {
        query.value = ""
        typeId.value = null
        tagId.value = null
    }
}
```

Note that `distinctUntilChanged()` stops a repeat query when the debounced text settles on a value the filters already hold. The `@OptIn` line acknowledges that `debounce` and `flatMapLatest` are marked as preview or experimental in the coroutines library; check which annotations your version asks for. Also, the removed "TEMPORARY" sample code from Chapter 6 is gone for good here.

Step 4: Passing the Title to the Form

The "Add" button in the not-found state must open the form with the searched text as the title. Give the route an optional starting title and use it in the form view model:

```
// Routes.kt
@Serializable data class FormRoute(val itemId: Long = 0L, val initialTitle: String = "")

// FormViewModel.kt: read the whole route once, and use it for the starting state
private val route = savedStateHandle.toRoute<FormRoute>()
private val itemId: Long = route.itemId
private val _state = MutableStateFlow(
    FormUiState(isEdit = itemId != 0L, title = route.initialTitle)
)
```

In the NavHost the list destination changes its add callback to take the title: `onAddClick = { title -> navController.navigate(FormRoute(initialTitle = title)) }`. The edit load in `FormViewModel` overwrites the title with the stored one, so `initialTitle` only affects a new item, as intended.

Step 5: The Screen

`ListScreen` now passes `onQueryChange`, `onTagSelected` and `onClearFilters` from the view model, and `onAddClick` becomes `(String) -> Unit`. Replace `ListContent` with the version below; `TypeFilterRow` and `ItemRow` from Chapter 4 stay as they are, and a second filter row is added for tags. Remember to update the preview or call sites for the new parameters.

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ListContent(
```

```
    state: ListUiState,
    onQueryChange: (String) -> Unit,
    onTypeSelected: (Long?) -> Unit,
    onTagSelected: (Long) -> Unit,
    onClearFilters: () -> Unit,
    onAddClick: (String) -> Unit,
    onItemClick: (Long) -> Unit
) {
    Scaffold(
        topBar = { TopAppBar(title = { Text("My Catalogue") }) },
        floatingActionButton = {
            FloatingActionButton(onClick = { onAddClick("") }) {
                Icon(Icons.Filled.Add, contentDescription = "Add item")
            }
        }
    ) { innerPadding ->
        Column(Modifier.padding(innerPadding)) {
            OutlinedTextField(
                value = state.query,
                onValueChange = onQueryChange,
                label = { Text("Search title or creator") },
                leadingIcon = { Icon(Icons.Filled.Search, contentDescription = null) },
                trailingIcon = {
                    if (state.hasQuery) {
                        IconButton(onClick = { onQueryChange("") }) {
                            Icon(Icons.Filled.Close, contentDescription = "Clear search"
                        }
                    }
                },
                singleLine = true,
                modifier = Modifier.fillMaxWidth().padding(horizontal = 16.dp, vertical =
            )
            TypeFilterRow(state, onTypeSelected)
            if (state.tags.isNotEmpty()) TagFilterRow(state, onTagSelected)

            when {
                state.isLoading -> Unit
                state.items.isEmpty() -> NoResults(state, onClearFilters, onAddClick)
                else -> {
                    if (state.hasQuery) {
                        Text(
                            text = "You already have ${state.items.size} matching item${
                            style = MaterialTheme.typography.labelLarge,
                            color = MaterialTheme.colorScheme.primary,
                            modifier = Modifier.padding(horizontal = 16.dp, vertical = 8
```

```
                        )
                    }
                    LazyColumn(
                        contentPadding = PaddingValues(16.dp),
                        verticalArrangement = Arrangement.spacedBy(8.dp)
                    ) {
                        items(state.items, key = { it.item.id }) { row ->
                            ItemRow(row, onClick = { onItemClick(row.item.id) })
                        }
                    }
                }
            }
        }
    }
}

@Composable
private fun TagFilterRow(state: ListUiState, onTagSelected: (Long) -> Unit) {
    LazyRow(
        contentPadding = PaddingValues(horizontal = 16.dp),
        horizontalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(state.tags, key = { it.id }) { tag ->
            FilterChip(
                selected = state.selectedTagId == tag.id,
                onClick = { onTagSelected(tag.id) },
                label = { Text("#${tag.name}") }
            )
        }
    }
}

/** The "do I already own this?" answer when nothing matches. */
@Composable
private fun NoResults(state: ListUiState, onClearFilters: () -> Unit, onAddClick: (Strin
    Column(
        Modifier.fillMaxSize().padding(32.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        when {
            state.totalCount == 0 -> Text("Your catalogue is empty. Tap + to add your fi
            state.hasChipFilter -> {
                Text("Nothing matches the current filters.")
                TextButton(onClick = onClearFilters) { Text("Clear filters") }
```

```
                }
                else -> {
                    Text("Not in your catalogue.", style = MaterialTheme.typography.titleMed
                    Spacer(Modifier.height(12.dp))
                    Button(onClick = { onAddClick(state.query.trim()) }) { Text("Add \"${sta
                }
            }
        }
    }
}
```

The `when` in `NoResults` is the "do I own this?" logic, in order: no items at all; nothing under the *chip* filters (the answer is not "you don't own it", it is "not within these filters", so offer to clear them); otherwise the search text found nothing, which really does mean it is not in the catalogue, so offer to add it. That distinction is why the chip filters and the text query are tracked separately. Add `Icons.Filled.Search` and the other imports the IDE asks for.

Try It

Add a few items with different types and tags. Type part of a title: matching items appear after a short pause and the heading says how many you own. Type something you do not have: "Not in your catalogue" appears with an Add button; tap it and the form opens with the text as the title. Select a tag chip, then a type chip, and see them combine; tap the tag chip again to turn it off. Finally search for `100%` and `a_b` to check that the escaping does not treat `%` and `_` as wildcards.

> **COMPARE WITH THE IOS COURSE**
>
> In *Personal Catalogue: iOS (Swift & SwiftUI)* Chapter 8, a `.searchable` modifier and a `#Predicate` in the `@Query` do the same job, and SwiftData rebuilds the query as the search text changes. The "do I own this?" answer and the add-with-title shortcut are the same design. The debounce and `flatMapLatest` plumbing mostly disappears there because SwiftUI re-runs the query itself.

Challenges

EXERCISE 1

Make all the changes and run the app. With at least six items (two of them tagged "sci-fi", one a CD), record what appears for each of: the search "dun"; the search "xyz"; the tag "sci-fi" alone; the tag "sci-fi" plus the type CD; and "xyz" with the tag "sci-fi" selected. Say which message each case shows and why they differ.

View solution

EXERCISE 2

Extend the search so it also matches the `notes` field. Say what you changed, whether the result is still sensible for a "do I own this?" check, and describe one way to make the search case-insensitive for accented letters (such as É and é) despite the ASCII-only behaviour of `LIKE`.

View solution

EXERCISE 3

Explain, with a concrete example of timings, what could go wrong if the view model used `map` and collected each query's flow separately instead of `flatMapLatest`. Then say what the user would see if `likePattern` did not escape `%` and they searched for "100%".

View solution

CHAPTER 8 QUICK REFERENCE

- One query, three optional conditions: `(:x IS NULL OR col = :x)`; text: `(:pattern = '' OR title LIKE :pattern ESCAPE '\' OR creator LIKE :pattern ESCAPE '\')`
- Tag filter: `id IN (SELECT itemId FROM item_tag WHERE tagId = :tagId)`
- `LIKE`: `%` any run, `_` any one; case-insensitive for ASCII only; escape `\` first, then `%` and `_`
- `query.debounce(250)` waits for a pause; `flatMapLatest { repository.search(...) }` cancels the previous query
- Keep the displayed text and the searched (debounced) text as separate flows
- Empty-result logic: no items → "empty catalogue"; chip filter active → "clear filters"; text only → "Not in your catalogue" + Add with the title

- `FormRoute(itemId, initialTitle)` pre-fills the form for a new item

NOT YET VERIFIED

Checked (two web searches on 2026-11-19): from the official `Flow` reference, that `debounce` drops values followed by newer ones within the timeout and always emits the latest, and that `flatMapLatest` maps each value to a flow and switches to the newest; from SQLite's documentation, that `LIKE` uses `%` and `_`, is case-insensitive for ASCII only, and that `ESCAPE` takes a single (ASCII) character that makes the next `%`, `_` or itself literal.

Not checked, from general knowledge: that Room accepts nullable query parameters and the `:param IS NULL OR` idiom; that a backslash inside a Kotlin raw string reaches SQLite unchanged; that Room's compile-time SQL check accepts `ESCAPE '\'`; the exact opt-in annotations for `debounce` and `flatMapLatest` in your coroutines version; the `combine` overloads with a constructor reference; `Icons.Filled.Search`; and all code, none of which was compiled. The forum threads in the search results were informal and I did not treat them as sources.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): kotlinx.coroutines Flow reference; SQLite, SQL Language Expressions (LIKE and ESCAPE); SQLite User Forum: case insensitive like and accented letters (a user question, used only as an illustration of the accent limit). Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: Android (Kotlin & Compose) 9 adds the duplicate warning when saving an item with the same title and creator, a light and dark theme check, accessibility labels, and proper error states.

Personal Catalogue: Android (Kotlin & Compose)

Course 1 · Chapter 9 · Warnings, Themes, Accessibility and Error States

The app now does everything the spec asks of it except the last item (backup, in Chapter 10). This chapter is the difference between an app that works and one you are happy to rely on. You will add a **duplicate warning** (the spec says "a warning, never a block"), check that the app looks right in **light and dark themes**, make it usable with a screen reader and large text, and replace the places where something can silently go wrong with proper **loading, error and not-found states**.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Nothing was compiled or run** (no Android Studio here), and **nothing was tested with TalkBack or in dark mode**: accessibility and appearance can only really be judged on a device, so treat this chapter's checks as a procedure for you to carry out. Three searches were made: the accessibility pages confirmed the 48dp touch-target rule, `contentDescription = null` for decorative icons and `clearAndSetSemantics` in Google's codelab; the Material 3 pages confirmed the theme pattern. Room's `COLLATE NOCASE` in a query and some semantics functions are from my memory.

Part 1: The Duplicate Warning

The spec's rule: when the user saves an item with the **same title and creator** (ignoring case) as one that exists, warn, but never block. A person can legitimately own two copies of a book, so the app's job is to ask, not to forbid. This is the reason Chapter 2 did *not* use a unique index (Exercise 2 there).

APPROACH	RESULT	USED HERE?
Unique index on title + creator	Database rejects the second copy; the user is blocked	No: contradicts the spec
Check in code, then ask	A query before saving; a dialog with "Save anyway"	Yes
Check only at the end of typing	Early feedback, but more moving parts	Left as a later improvement

The query

Add to `ItemDao`. `COLLATE NOCASE` makes the comparison ignore case (for ASCII letters, the same limit as Chapter 8's search), and `id != :excludeId` stops an item being reported as a duplicate of *itself* when you edit it.

```
@Query(
    """
    SELECT * FROM item
    WHERE title = :title COLLATE NOCASE
      AND creator = :creator COLLATE NOCASE
      AND id != :excludeId
    """
)
suspend fun findDuplicates(title: String, creator: String, excludeId: Long): List<Item>
```

In the repository, trim the values the same way the form does before saving, so "Dune " and "Dune" count as the same:

```
suspend fun findDuplicates(title: String, creator: String, excludeId: Long): List<Item> =
    itemDao.findDuplicates(title.trim(), creator.trim(), excludeId)
```

The form's save flow

Saving becomes a small decision tree: invalid, so show errors; duplicate found, so show the warning and wait; otherwise write. Two choices from Chapter 6 stay (the `isSaving` guard against double-taps, and carrying `createdAt` through an edit), and a third is new: **every write is wrapped so a failure is reported instead of lost** (Part 4 shows the other half). Add two fields to `FormUiState`:

```
val duplicateCount: Int = 0,        // > 0 means: show the duplicate warning
val saveFailed: Boolean = false,    // true means: show the "could not save" message once
```

Replace `save` in `FormViewModel` with the following, and add the three small functions after it. (Import `kotlinx.coroutines.CancellationException`.)

```
fun save(onSaved: () -> Unit) {
    val s = _state.value
    if (s.isSaving) return
    if (!s.isValid) {
        _state.update { it.copy(showErrors = true) }
        return
    }
    _state.update { it.copy(isSaving = true) }
    viewModelScope.launch {
        guarded {
            val duplicates = repository.findDuplicates(s.title, s.creator, excludeId = i
            if (duplicates.isNotEmpty()) {
                _state.update { it.copy(isSaving = false, duplicateCount = duplicates.si
            } else {
                write(s)
                onSaved()
            }
        }
    }
}

/** The user saw the warning and chose to go ahead. */
fun saveAnyway(onSaved: () -> Unit) {
    val s = _state.value
    if (s.isSaving) return
    _state.update { it.copy(isSaving = true, duplicateCount = 0) }
    viewModelScope.launch { guarded { write(s); onSaved() } }
}

fun dismissDuplicateWarning() = _state.update { it.copy(duplicateCount = 0) }
fun errorShown() = _state.update { it.copy(saveFailed = false) }

private suspend fun write(s: FormUiState) {
    repository.saveItemWithTagNames(
        Item(
            id = itemId,
            title = s.title.trim(),
            creator = s.creator.trim(),
            formatDetail = s.formatDetail.trim(),
            releaseYear = s.yearText.toIntOrNull(),
            notes = s.notes.trim(),
            createdAt = originalCreatedAt,
            itemTypeId = s.typeId!!
        ),
```

```
            tagNames = s.tagNames
        )
    }

    /** Runs a save step; reports a failure instead of crashing or failing silently. */
    private suspend fun guarded(block: suspend () -> Unit) {
        try {
            block()
        } catch (e: CancellationException) {
            throw e                          // never swallow cancellation
        } catch (e: Exception) {
            _state.update { it.copy(isSaving = false, saveFailed = true) }
        }
    }
```

Rethrowing `CancellationException` matters: coroutines use it to cancel work (for example when the user leaves the screen), and a blanket `catch (e: Exception)` that swallows it would break that. Catching the general `Exception` is acceptable here because the only thing we do is tell the user; a real app would also log it.

The dialog and the message in the form

Give `FormContent` three more parameters (`onSaveAnyway`, `onDismissDuplicate`, `onErrorShown`), pass them from `FormScreen` as `{ viewModel.saveAnyway(onDone) }`, `viewModel::dismissDuplicateWarning` and `viewModel::errorShown`, and add the following. The failure message uses a **snackbar**: a brief message at the bottom of the screen that does not block anything.

```
    // At the top of FormContent:
    val snackbarHostState = remember { SnackbarHostState() }
    LaunchedEffect(state.saveFailed) {
        if (state.saveFailed) {
            snackbarHostState.showSnackbar("Couldn't save. Please try again.")
            onErrorShown()
        }
    }

    // In the Scaffold( ... ) call, add:
    snackbarHost = { SnackbarHost(snackbarHostState) },

    // After the Scaffold, still inside FormContent:
    if (state.duplicateCount > 0) {
```

```
    AlertDialog(
        onDismissRequest = onDismissDuplicate,
        title = { Text("You may already own this") },
        text = {
            Text(
                "You already have ${state.duplicateCount} item${if (state.duplicateCount
                "called \"${state.title.trim()}\" by the same creator. Save another copy
            )
        },
        confirmButton = { TextButton(onClick = onSaveAnyway) { Text("Save anyway") } },
        dismissButton = { TextButton(onClick = onDismissDuplicate) { Text("Go back") } }
    )
}
```

`LaunchedEffect(state.saveFailed)` runs its block when the key changes: it shows the snackbar once, then tells the view model the error was shown so it is not shown again after rotation. That "show once, then clear" step is the standard way to handle a one-off message that lives in state.

Part 2: Light and Dark Theme

The wizard generated `ui/theme/Theme.kt` in Chapter 1, and it already supports both themes. Google's Material 3 guide says to use `isSystemInDarkTheme()` to choose between a light and a dark scheme, and its codelab's theme function takes a `darkTheme` parameter that defaults to it. The generated file normally looks like this in outline (yours will differ in detail):

```
@Composable
fun CatalogueTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    dynamicColor: Boolean = true,      // colours from the wallpaper on Android 12+
    content: @Composable () -> Unit
) {
    val colorScheme = when {
        dynamicColor && Build.VERSION.SDK_INT >= Build.VERSION_CODES.S -> {
            val context = LocalContext.current
            if (darkTheme) dynamicDarkColorScheme(context) else dynamicLightColorScheme(
        }
        darkTheme -> DarkColorScheme       // built with darkColorScheme(...)
        else -> LightColorScheme          // built with lightColorScheme(...)
    }
```

```
        MaterialTheme(colorScheme = colorScheme, typography = Typography, content = content)
    }
```

So you have very little to *write*; the work is to **check**. The rule that makes dark mode work is that screens use roles from the theme, such as `MaterialTheme.colorScheme.primary` and `onSurface`, and never a fixed colour. The code in this course has followed that rule, so the checks are:

1. **Search for hard-coded colours.** Use **Edit > Find > Find in Files** for `Color(` and `Color.` in `ui/` outside the theme folder. Any hit is a colour that will not change with the theme; replace it with a `colorScheme` role.
2. **Try both themes.** On the emulator open *Settings* > *Display* > *Dark theme* and switch it while the app is open. Look at every screen: list, detail, form (with an error and with the duplicate dialog open) and the empty states.
3. **Preview both.** Add a second `@Preview` with `uiMode = Configuration.UI_MODE_NIGHT_YES` next to each screen's preview, so you can compare them in the IDE.
4. **Dynamic colour and contrast.** With dynamic colour on, the palette depends on the user's wallpaper, so you cannot check every combination; sticking to theme roles (text on its matching `on...` colour) is what keeps text readable.

The same parameter lets you force a theme regardless of the system setting, which is how an in-app "Light / Dark / System" choice is built; this course leaves that out because the spec only asks that both themes look right.

Part 3: Accessibility

Accessibility on Android mostly means three things: a screen reader (TalkBack) must be able to say what each element is and does; touch targets must be large enough; and text must survive being enlarged. Google's guidance supports each of these, and Compose does a lot for free: Material components such as `Button` and `IconButton` already enforce the 48dp minimum touch target. Your job is the gaps.

ISSUE	WHERE IN THIS APP	FIX
Icon with no description	Back, edit, delete, add, clear, remove-tag icons	Already given a `contentDescription` in Chapters 4-8; check each one still reads naturally
Decorative icon	The search field's leading magnifier	`contentDescription = null`, as Chapter 8 wrote: Google's guidance says that tells the framework it has no action or state
A row read as many fragments	List rows (title, creator, type, tags)	Make the whole row one clickable unit with a clear action label
A clickable thing smaller than 48dp	Nothing now, but any future custom control	`Modifier.minimumInteractiveComponentSize()`
Colour as the only signal	Form errors	Already text as well as red (`supportingText`), so no change
Large text	Rows and forms	Test at the largest font size; keep `maxLines` and scrolling

The row fix is one change in Chapter 4's `ItemRow`. Giving the click an `onClickLabel` makes TalkBack announce "*Open details*" as the action instead of just "double-tap to activate". A clickable container already groups its children into one element for the screen reader; the label tells the user what the tap does.

```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .clickable(onClickLabel = "Open details", onClick = onClick)
) {
    // ...unchanged...
}
```

Google's accessibility codelab also shows `clearAndSetSemantics`, which *replaces* everything a TalkBack user would hear for an element with text you write yourself. It is powerful but easy to misuse: you take over responsibility for the whole description. Prefer the clickable label above, and reach for `clearAndSetSemantics` only when the automatic reading is genuinely wrong.

Now **test**. Turn on TalkBack (*Settings* > *Accessibility* > *TalkBack*), swipe through the list, detail and form screens, and listen: every control should announce what it is, and nothing should be read

twice or not at all. Then set *Settings* > *Display* > *Font size* to the largest, and check nothing is cut off or unreachable. Google's "Accessibility Scanner" app (and the Compose accessibility testing guide) can flag missing labels and small targets automatically.

Part 4: Loading, Error and Not-Found States

A screen should always be in one of a few explicit states, never blank without explanation. Here is the audit for this app:

SCREEN	STATE	BEFORE THIS CHAPTER	AFTER
List	Loading	A blank screen for a moment	A progress indicator
List	Empty, or no match	Handled (Chapters 4 and 8)	No change
Detail	Loading, not found	Handled (Chapter 5)	No change
Form	Save fails	Crash, or nothing	A snackbar, and the form stays as it was
Detail	Delete fails	Crash, or nothing	Exercise 3

Form errors are done (Part 1). For the list's loading state, change one line in `ListContent`'s `when` from Chapter 8: `state.isLoading -> Unit` becomes

```
state.isLoading -> Box(Modifier.fillMaxSize(), contentAlignment = Alignment.Center) {
    CircularProgressIndicator()
}
```

This is the smallest useful version. On a local database loading is nearly instant, so the indicator is rarely seen; it matters in the one case where it counts, the first launch after install while the database is being created and seeded.

A LIMIT: HARD-CODED TEXT

All the screen text in this course is written directly in the Kotlin code. A shipped app puts it in `res/values/strings.xml` and reads it with `stringResource(R.string.x)`, which is what allows translation and lets accessibility tools and tests refer to strings by name. That is a mechanical change this course leaves out to keep the code readable; *Android Development: Production & Publishing* covers release preparation.

COMPARE WITH THE IOS COURSE

In *Personal Catalogue: iOS (Swift & SwiftUI)* Chapter 9 the same four jobs appear. A confirmation dialog warns about duplicates; SwiftUI's semantic colours follow light and dark automatically; VoiceOver labels use `.accessibilityLabel`; and a failed save is shown with an alert. The difference is that on iOS much of this is automatic, whereas on Android you check more of it yourself.

Challenges

EXERCISE 1

Add the duplicate check, the new state, the save flow and the dialog. Test and record the result of: (a) adding "Dune" by "Frank Herbert" when "dune" by "frank herbert" exists; (b) editing that existing item and saving it unchanged; (c) adding "Dune" by "Someone Else"; (d) choosing "Save anyway" in case (a).

View solution

EXERCISE 2

Carry out the accessibility and theme checks on a device or emulator: dark theme, TalkBack and the largest font size. Write down at least three real findings (something unclear, cut off or awkward) and fix one of them.

View solution

EXERCISE 3

Chapter 5's `DetailViewModel.delete` has no error handling. Add it: report a failed delete to the user and keep them on the screen. Say where the state and the message should live.

View solution

CHAPTER 9 QUICK REFERENCE

- Duplicate rule: same title and creator, case-insensitive: `title = :title COLLATE NOCASE AND creator = :creator COLLATE NOCASE AND id != :excludeId`; a warning, never a block

- Save flow: invalid → show errors; duplicate → dialog with "Save anyway"; otherwise write; any failure → snackbar
- Wrap writes in a function that rethrows `CancellationException` and reports other exceptions
- One-off messages in state: show in `LaunchedEffect(flag)`, then clear the flag
- Theme: `isSystemInDarkTheme()`, `lightColorScheme` / `darkColorScheme`, dynamic colour on Android 12+; use theme roles, never fixed colours
- Accessibility: meaningful `contentDescription`, `null` for decorative, `onClickLabel` on clickable rows, 48dp targets, test with TalkBack and the largest font
- Every screen has explicit loading, empty, error and not-found states

NOT YET VERIFIED

Checked (three web searches on 2026-11-19): that Google's guidance sets 48dp as the minimum touch target and that Material components such as `Button` and `IconButton` enforce it; that `minimumInteractiveComponentSize` exists for custom controls; that `contentDescription = null` marks decorative elements; that `clearAndSetSemantics` appears in Google's accessibility codelab; that `isSystemInDarkTheme()` chooses light or dark schemes and that a `darkTheme` parameter can override it; that dynamic colour needs Android 12 (API 31) and a fallback.
Not checked, from general knowledge: `COLLATE NOCASE` on query parameters in a Room `@Query`; `clickable(onClickLabel = ...)` and its effect on TalkBack; that a clickable `Card` merges its children's semantics; that `FilterChip` announces its selected state; `SnackbarHost` and `LaunchedEffect` usage details; the layout of your generated `Theme.kt`; the Accessibility Scanner app; and all code, none of which was compiled. **No screen reader, dark-mode or large-font testing was done.**

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Compose accessibility API defaults; Accessibility in Jetpack Compose (codelab); Make apps more accessible; Testing Compose accessibility; Material Design 3 in Compose; Theming in Compose with Material 3 (codelab); isSystemInDarkTheme API reference. Only titles and short summaries in the search results were used.

COMING NEXT

Personal Catalogue: Android (Kotlin & Compose) 10 is the last chapter: exporting the whole catalogue to a JSON file and importing it back with the Storage Access Framework, tests for the repository and

the view models, and a roadmap (barcode scanning with CameraX and ML Kit), compared with the iOS course. The combined course PDF is built when it lands.

Personal Catalogue: Android (Kotlin & Compose)

Course 1 · Chapter 10 · JSON Backup, Tests and the Road Ahead

The last item on the spec's essential list is **backup**, and it matters more than it sounds. The catalogue lives only on one phone, in a database no one else can read. Lose the phone, or clear the app's storage, and years of careful cataloguing are gone. This chapter lets the user **export everything to a JSON file** they choose the location of, and **import** it back. Then it adds tests, so the rules you have built are protected when you change things later, and ends with where the app could go next.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> **Nothing was compiled or run** (no Android Studio here), and no test in this chapter has been executed. Two searches were made. The first confirmed the Storage Access Framework rules (the user picks the file with `ACTION_CREATE_DOCUMENT` or `ACTION_OPEN_DOCUMENT`, which gives your app access to just that file) and the Compose launcher pattern, but **did not show `openOutputStream` used with those results**, so that call is from memory. The second confirmed the in-memory-database approach to testing Room, but **the Room testing page describes newer setups (`BundledSQLiteDriver`, JVM tests) that I have not followed**: the tests here use the long-established instrumented pattern, so check the page for your Room version.

Where We Are

SPEC ITEM	BUILT IN
1. Add, edit, delete, labels per type	Chapters 5 and 6
2. List, newest first, type filter	Chapters 4 and 8
3. Detail screen	Chapter 5
4. Managed tags	Chapters 2, 3 and 7
5. Search and tag filter	Chapter 8
6. Duplicate warning	Chapter 9
7. Local storage, offline	Chapters 2 and 3

SPEC ITEM	BUILT IN
8. Light and dark theme, empty states, accessibility	Chapters 4, 8 and 9
9. Backup: export and import JSON	**This chapter**

Part 1: A Backup Format

A backup file must outlive the database it came from, so it must not contain database ids. Row ids are an accident of one install; if you restore into a fresh database the ids differ. Instead the file stores an item's type and tags *by name*, and the importer looks them up again. It also carries a `version` number, so a future app that adds fields (Part 5) can still read old files, and can refuse files from a future version it does not understand. You already added the Kotlin serialization plugin and library in Chapter 5; the same library encodes and decodes JSON.

```
package com.example.catalogue.backup

import com.example.catalogue.data.ItemWithDetails
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json

@Serializable
data class BackupFile(
    val version: Int = CURRENT_VERSION,
    val exportedAt: Long,
    val items: List<BackupItem>
) {
    companion object { const val CURRENT_VERSION = 1 }
}

@Serializable
data class BackupItem(
    val title: String,
    val creator: String = "",
    val formatDetail: String = "",
    val releaseYear: Int? = null,
    val notes: String = "",
    val createdAt: Long,
    val type: String,                       // "Book", "CD", ... by name, not id
    val tags: List<String> = emptyList()
)
```

```
val backupJson = Json {
    prettyPrint = true            // a human can open and read the file
    encodeDefaults = true
    ignoreUnknownKeys = true      // tolerate fields added by a newer version
}

fun ItemWithDetails.toBackup() = BackupItem(
    title = item.title,
    creator = item.creator,
    formatDetail = item.formatDetail,
    releaseYear = item.releaseYear,
    notes = item.notes,
    createdAt = item.createdAt,
    type = type.name,
    tags = tags.map { it.name }.sorted()
)
```

The exported file looks like ordinary JSON, which is the point: it can be read in a text editor, kept in cloud storage, or fixed by hand. Compare the web catalogues' database dumps; this is the same idea in a portable form.

Part 2: Export and Import in the Repository

Two new read queries are needed, and one to detect an item that is already present:

```
// ItemTypeDao
@Query("SELECT * FROM item_type ORDER BY id")
suspend fun getAll(): List<ItemType>

// ItemDao
@Transaction
@Query("SELECT * FROM item ORDER BY createdAt")
suspend fun getAll(): List<ItemWithDetails>

@Query(
    """
    SELECT COUNT(*) FROM item
    WHERE title = :title AND creator = :creator AND itemTypeId = :typeId AND createdAt =
    """
```

```
    )
    suspend fun countExact(title: String, creator: String, typeId: Long, createdAt: Long): I
```

Then two repository functions. Three design decisions are worth reading before the code:

- **Import merges; it does not replace.** Existing items are never touched. An incoming item that matches an existing one on title, creator, type *and creation time* is skipped, so importing the same file twice does not double the catalogue. (A deliberate duplicate, Chapter 9's "I own two copies", has a different creation time, so it survives a round trip.)
- **Import is all-or-nothing.** It runs in one transaction; if any item is bad, an exception rolls everything back, and the user is never left with a half-imported catalogue.
- **Unknown types are an error, not a guess.** The four types are a fixed list, so an item naming another type means the file is not from this app.

```
    data class ImportResult(val added: Int, val skipped: Int)

    suspend fun exportBackup(): String {
        val all = itemDao.getAll()
        return backupJson.encodeToString(
            BackupFile(exportedAt = System.currentTimeMillis(), items = all.map { it.toBacku
        )
    }

    /** Throws SerializationException for bad JSON, IllegalArgumentException for a bad backu
    suspend fun importBackup(json: String): ImportResult {
        val file = backupJson.decodeFromString<BackupFile>(json)
        require(file.version <= BackupFile.CURRENT_VERSION) { "This backup was made by a new
        ensureSeeded()

        return db.withTransaction {
            val types = itemTypeDao.getAll().associateBy { it.name.lowercase() }
            var added = 0
            var skipped = 0
            for (b in file.items) {
                val type = types[b.type.lowercase()]
                    ?: throw IllegalArgumentException("Unknown item type \"${b.type}\".")
                val title = b.title.trim()
                val creator = b.creator.trim()
                if (title.isEmpty() || itemDao.countExact(title, creator, type.id, b.created
                    skipped++
                    continue
```

```
                }
                val tagIds = b.tags.map { getOrCreateTag(it).id }.toSet()
                writeItem(
                    Item(
                        title = title, creator = creator, formatDetail = b.formatDetail.trim
                        releaseYear = b.releaseYear, notes = b.notes.trim(),
                        createdAt = b.createdAt, itemTypeId = type.id
                    ),
                    tagIds
                )
                added++
            }
            ImportResult(added, skipped)
        }
    }
```

The decoding happens *before* the transaction starts, so a file that is not JSON at all fails fast without touching the database. Everything after it uses the functions you already trust: `getOrCreateTag` (Chapter 7) keeps tag names unique, and `writeItem` writes the item and its links.

Part 3: The Storage Access Framework

How does the app get a place to write the file, without asking for broad storage permission? Through the **Storage Access Framework**. Per Google's documentation, apps go through `ACTION_CREATE_DOCUMENT` (save a single file) or `ACTION_OPEN_DOCUMENT` (open one), which require the user to pick the location or the file themselves; your app is then granted a **URI permission** for exactly that one document. The practical result is that you need *no storage permission in the manifest*, and the user keeps full control of where their backup goes (device storage, a cloud drive, an SD card).

STEP	EXPORT	IMPORT
Contract	`ActivityResultContracts.CreateDocument("application/json")`	`ActivityResultContracts.OpenD`
Launch	`launcher.launch("catalogue-backup.json")` (suggested name)	`launcher.launch(arrayOf("appl ...))` (accepted types)

STEP	EXPORT	IMPORT
Your code	Open an output stream on the Uri and write the JSON	Open an input stream on the Uri text, import it
Web equivalent	A "Download backup" link	A file upload form

The view model

The view model needs the application `Context` to reach the `ContentResolver` (the system service that turns a `Uri` into a stream); Hilt supplies it with `@ApplicationContext`, which is safe to hold because it lives as long as the app. Reading and writing files is slow, so it runs on `Dispatchers.IO`.

```
package com.example.catalogue.backup

import android.content.Context
import android.net.Uri
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.example.catalogue.data.CatalogueRepository
import dagger.hilt.android.lifecycle.HiltViewModel
import dagger.hilt.android.qualifiers.ApplicationContext
import javax.inject.Inject
import kotlinx.coroutines.CancellationException
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.update
import kotlinx.coroutines.launch
import kotlinx.coroutines.withContext
import kotlinx.serialization.SerializationException

data class BackupUiState(val working: Boolean = false, val message: String? = null)

@HiltViewModel
class BackupViewModel @Inject constructor(
    @ApplicationContext private val context: Context,
    private val repository: CatalogueRepository
) : ViewModel() {
```

```
    private val _state = MutableStateFlow(BackupUiState())
    val uiState: StateFlow<BackupUiState> = _state.asStateFlow()

    fun exportTo(uri: Uri) = launchWork {
        val json = repository.exportBackup()
        withContext(Dispatchers.IO) {
            context.contentResolver.openOutputStream(uri, "wt")?.use { it.write(json.toB
                ?: error("Could not open the file for writing.")
        }
        "Backup saved."
    }

    fun importFrom(uri: Uri) = launchWork {
        val text = withContext(Dispatchers.IO) {
            context.contentResolver.openInputStream(uri)?.bufferedReader()?.use { it.rea
                ?: error("Could not open the file for reading.")
        }
        val result = repository.importBackup(text)
        "Imported ${result.added} item${if (result.added == 1) "" else "s"}; ${result.sk
    }

    fun messageShown() = _state.update { it.copy(message = null) }

    private fun launchWork(block: suspend () -> String) {
        if (_state.value.working) return
        _state.update { it.copy(working = true, message = null) }
        viewModelScope.launch {
            val message = try {
                block()
            } catch (e: CancellationException) {
                throw e
            } catch (e: SerializationException) {
                "That file is not a catalogue backup."
            } catch (e: IllegalArgumentException) {
                e.message ?: "That backup could not be imported."
            } catch (e: Exception) {
                "Something went wrong: ${e.message ?: "unknown error"}"
            }
            _state.update { it.copy(working = false, message = message) }
        }
    }
}
```

The `catch` order goes from most specific to most general, and the final message is the app's Chapter 9 philosophy again: every failure ends in a sentence the user can read, never a silent crash. Note that `"wt"` asks to truncate the file when writing, so an overwritten backup does not keep stale trailing bytes.

The screen

`rememberLauncherForActivityResult` is the Compose way to start a "go and pick something" action and receive the result; Google's documentation says it handles registering and unregistering the launcher for you. Each launcher is given a contract, and its callback receives the chosen `Uri`, or `null` if the user backed out.

```
package com.example.catalogue.backup

import androidx.activity.compose.rememberLauncherForActivityResult
import androidx.activity.result.contract.ActivityResultContracts
import androidx.compose.foundation.layout.*
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.compose.collectAsStateWithLifecycle

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun BackupScreen(onBack: () -> Unit, viewModel: BackupViewModel = hiltViewModel()) {
    val state by viewModel.uiState.collectAsStateWithLifecycle()

    val exportLauncher = rememberLauncherForActivityResult(
        ActivityResultContracts.CreateDocument("application/json")
    ) { uri -> if (uri != null) viewModel.exportTo(uri) }

    val importLauncher = rememberLauncherForActivityResult(
        ActivityResultContracts.OpenDocument()
    ) { uri -> if (uri != null) viewModel.importFrom(uri) }

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text("Backup and restore") },
```

```
                navigationIcon = {
                    IconButton(onClick = onBack) {
                        Icon(Icons.AutoMirrored.Filled.ArrowBack, contentDescription = "
                    }
                }
            )
        }
    ) { innerPadding ->
        Column(
            Modifier.padding(innerPadding).padding(16.dp),
            verticalArrangement = Arrangement.spacedBy(16.dp)
        ) {
            Text(
                "Save your whole catalogue to a file you choose, or restore from one. " +
                "Restoring adds items; it never deletes or changes what is already here.
            )
            Button(
                onClick = { exportLauncher.launch("catalogue-backup.json") },
                enabled = !state.working,
                modifier = Modifier.fillMaxWidth()
            ) { Text("Export backup") }
            OutlinedButton(
                onClick = { importLauncher.launch(arrayOf("application/json", "text/plai
                enabled = !state.working,
                modifier = Modifier.fillMaxWidth()
            ) { Text("Import backup") }

            if (state.working) LinearProgressIndicator(Modifier.fillMaxWidth())
            state.message?.let { Text(it, style = MaterialTheme.typography.bodyLarge) }
        }
    }
}
```

The import launcher lists several accepted types because file managers and cloud drives are inconsistent about labelling a `.json` file; being generous avoids a greyed-out file. The message shows in place; a snackbar would also do, as in Chapter 9.

Getting to it

Add `@Serializable object BackupRoute` to `Routes.kt`, a `composable<BackupRoute> { BackupScreen(onBack = { navController.popBackStack() }) }` to the NavHost, a new

`onBackupClick: () -> Unit` parameter on `ListScreen` and `ListContent`, passed as `{ navController.navigate(BackupRoute) }`, and an action in the list's `TopAppBar`:

```
TopAppBar(
    title = { Text("My Catalogue") },
    actions = {
        IconButton(onClick = onBackupClick) {
            Icon(Icons.Filled.Settings, contentDescription = "Backup and restore")
        }
    }
)
```

Try it: export, then clear the app's storage (*Settings* > *Apps* > *Catalogue* > *Storage* > *Clear storage*), reopen it, and import. The items, types, tags, notes and creation dates should all be back. Open the exported file in a text editor to see the JSON for yourself. The file is **unencrypted plain text**, so the user should store it somewhere they trust; for a shopping-list-grade catalogue that is a reasonable trade for being readable and portable.

Part 4: Tests

You now have rules that are easy to break by accident: tag names must be unique ignoring case, search patterns must escape wildcards, a backup must survive a round trip. Tests turn those rules into things the computer checks for you. There are three layers, from cheapest to most involved, and you should write them in this order:

LAYER	TESTS	WHERE IT RUNS	FOLDER
Pure functions	`likePattern`, `normaliseTagName`, JSON round trip	Your computer, in milliseconds (a local test)	`app/src/test/...`
Repository on a real database	Saving with tags, cascades, duplicates, import	An emulator or device (an instrumented test), with an in-memory database	`app/src/androidTest/...`
View model	Search state, filters	Same as the repository, because it uses the real repository	`app/src/androidTest/...`

Add `testImplementation` for `kotlinx-coroutines-test` (and JUnit, which the wizard normally adds already), and `androidTestImplementation` for `kotlinx-coroutines-test` and the AndroidX test libraries; versions from the release pages, as always.

Layer 1: pure functions

```
package com.example.catalogue

import com.example.catalogue.backup.*
import com.example.catalogue.data.*
import org.junit.Assert.assertEquals
import org.junit.Test

class PureRulesTest {

    @Test fun emptySearchMeansNoPattern() {
        assertEquals("", likePattern("   "))
    }

    @Test fun searchWrapsInWildcards() {
        assertEquals("%dune%", likePattern(" dune "))
    }

    @Test fun wildcardsInTheQueryAreEscaped() {
        assertEquals("%100\\%%", likePattern("100%"))
        assertEquals("%a\\_b%", likePattern("a_b"))
        assertEquals("%a\\\\b%", likePattern("a\\b"))
    }

    @Test fun tagNamesAreTrimmedAndCollapsed() {
        assertEquals("retro games", normaliseTagName("  retro   games  "))
    }

    @Test fun backupSurvivesAJsonRoundTrip() {
        val original = BackupFile(
            exportedAt = 1_700_000_000_000,
            items = listOf(
                BackupItem(title = "Dune", creator = "Frank Herbert", releaseYear = 1965,
                    createdAt = 1_600_000_000_000, type = "Book", tags = listOf("classic"
                BackupItem(title = "Emma", createdAt = 1_600_000_100_000, type = "Book")
            )
        )
        val decoded = backupJson.decodeFromString<BackupFile>(backupJson.encodeToString(
        assertEquals(original, decoded)
```

```
    }
}
```

Notice what the escaping test shows: the Kotlin string `"%100\\%%"` is the four-character text `%100\%%`. These tests would have caught the "escape the backslash first" mistake discussed in Chapter 8 immediately. Run them with a right-click on the file and **Run 'PureRulesTest'**.

Layer 2: the repository on an in-memory database

An *in-memory* database lives only for the test and vanishes with it, so every test starts clean and nothing is left on the device. Google's Room testing page recommends this for isolation, and closing the database after each test. The pattern below is the long-established instrumented one, which needs an emulator or phone.

```
package com.example.catalogue

import androidx.room.Room
import androidx.test.core.app.ApplicationProvider
import com.example.catalogue.data.*
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.test.runTest
import org.junit.After
import org.junit.Assert.*
import org.junit.Before
import org.junit.Test

class RepositoryTest {
    private lateinit var db: AppDatabase
    private lateinit var repo: CatalogueRepository

    @Before fun setUp() {
        db = Room.inMemoryDatabaseBuilder(ApplicationProvider.getApplicationContext(), A
        repo = CatalogueRepository(db, db.itemTypeDao(), db.tagDao(), db.itemDao(), db.i
    }

    @After fun tearDown() { db.close() }

    private suspend fun bookTypeId(): Long {
        repo.ensureSeeded()
        return repo.itemTypes.first().first { it.name == "Book" }.id
    }
```

```
    @Test fun seedingIsIdempotent() = runTest {
        repo.ensureSeeded(); repo.ensureSeeded()
        assertEquals(listOf("Book", "CD", "DVD", "Blu-ray"), repo.itemTypes.first().map
    }

    @Test fun tagNamesAreUniqueIgnoringCase() = runTest {
        val a = repo.getOrCreateTag("Sci-Fi")
        val b = repo.getOrCreateTag("  sci-fi ")
        assertEquals(a.id, b.id)
        assertEquals(1, repo.tags.first().size)
    }

    @Test fun savingWithTagsLinksThemAndDeletingRemovesLinksOnly() = runTest {
        val id = repo.saveItemWithTagNames(Item(title = "Dune", itemTypeId = bookTypeId(
        val saved = repo.observeItem(id).first()!!
        assertEquals(setOf("sci-fi", "classic"), saved.tags.map { it.name }.toSet())

        repo.deleteItem(saved.item)
        assertNull(repo.observeItem(id).first())
        assertEquals(2, repo.tags.first().size)          // the tags themselves remain
    }

    @Test fun duplicateCheckIgnoresCaseAndTheItemItself() = runTest {
        val type = bookTypeId()
        val id = repo.saveItemWithTagNames(Item(title = "Dune", creator = "Frank Herbert
        assertEquals(1, repo.findDuplicates("dune", "FRANK HERBERT", excludeId = 0).size
        assertEquals(0, repo.findDuplicates("dune", "frank herbert", excludeId = id).siz
    }

    @Test fun importingTwiceDoesNotDuplicate() = runTest {
        val type = bookTypeId()
        repo.saveItemWithTagNames(Item(title = "Dune", itemTypeId = type, createdAt = 10
        val json = repo.exportBackup()

        val result = repo.importBackup(json)
        assertEquals(ImportResult(added = 0, skipped = 1), result)
        assertEquals(1, repo.items.first().size)
    }
}
```

The tests read like specifications of the app's own rules: seeding does not repeat, tag names are unique ignoring case, deleting an item removes its links but not the tags, a duplicate is detected

but an item is not its own duplicate, and importing twice adds nothing. If you ever change the data layer, these are the five things that must still be true.

Layer 3: a view model

The list view model is tested by giving it a repository on an in-memory database, then driving it the way the screen does. Two coroutine-testing facts apply, both in the library's documentation: view model code that launches coroutines needs the *main dispatcher* replaced with a test one, and the `debounce` delay is *virtual time*, advanced by the test rather than waited for. Treat the next test as the least certain code in this chapter: tests that mix a real database (running on its own threads) with virtual time are the ones most likely to need adjusting.

```
@OptIn(ExperimentalCoroutinesApi::class)
class ListViewModelTest {
    private lateinit var db: AppDatabase
    private lateinit var repo: CatalogueRepository

    @Before fun setUp() {
        Dispatchers.setMain(StandardTestDispatcher())
        db = Room.inMemoryDatabaseBuilder(ApplicationProvider.getApplicationContext(), A
        repo = CatalogueRepository(db, db.itemTypeDao(), db.tagDao(), db.itemDao(), db.i
    }

    @After fun tearDown() {
        db.close()
        Dispatchers.resetMain()
    }

    @Test fun searchNarrowsTheList() = runTest {
        repo.ensureSeeded()
        val book = repo.itemTypes.first { it.isNotEmpty() }.first { it.name == "Book" }.
        repo.saveItemWithTagNames(Item(title = "Dune", creator = "Frank Herbert", itemTy
        repo.saveItemWithTagNames(Item(title = "Emma", creator = "Jane Austen", itemType

        val vm = ListViewModel(repo)
        vm.onQueryChange("dun")

        val state = vm.uiState.first { it.query == "dun" && it.items.size == 1 }
        assertEquals("Dune", state.items.single().item.title)
        assertEquals(2, state.totalCount)
    }
}
```

Why `first { ... }` with a condition instead of a fixed wait? The state passes through several values on the way (loading, all items, then the filtered list); waiting for the one that matters makes the test independent of how long that takes. Add `import kotlinx.coroutines.test.*`, `kotlinx.coroutines.Dispatchers` and `kotlinx.coroutines.ExperimentalCoroutinesApi`.

Part 5: Where Next

The spec's "nice to have" list is the roadmap. For each, here is what actually changes, which tells you how big the job is:

FEATURE	WHAT IT NEEDS	TOUCHES
Barcode scanning	CameraX for the camera preview, ML Kit's barcode scanner to read the code, a camera permission request	A new screen; the form accepts a scanned value
Online lookup by ISBN or barcode	A network library (Retrofit or Ktor), an API such as Open Library or Google Books, the internet permission, parsing JSON, handling failures and no network	Repository gains a remote source; the form pre-fills from a result
ISBN, runtime, tracklist fields	New columns on the item (or type-specific tables), a Room **migration** from version 1 to 2, a backup format `version = 2`	Data model, form labels, detail screen, backup
Cover art	An image loading library (Coil), storing or caching images, a bigger backup story (images are not in the JSON)	Rows, detail, backup
Wish list, "lent to"	A status field on `Item` or a new table, filters for it	Data model, list filters, form
Cloud sync	Firebase (as in the web *Personal Catalogue: Firebase* course): authentication, conflict rules between devices	The repository becomes the sync boundary; most screens unchanged

Notice how little the screens would change in several rows. That is the payoff of the architecture you built: the repository is the one seam, so new data sources slot in behind it. The most important lesson for the data-model rows is the **migration**: once the app is installed on a phone with real data, you cannot just edit an entity; you raise the database version and tell Room how to upgrade. In outline (the signature differs between Room versions, so check the migration page for yours):

```
// Version 1 -> 2: add an optional ISBN column. Existing rows get NULL.
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL("ALTER TABLE item ADD COLUMN isbn TEXT")
    }
}

// ...and in the database builder:
Room.databaseBuilder(context, AppDatabase::class.java, "catalogue.db")
    .addMigrations(MIGRATION_1_2)
    .build()
```

Because the backup file stores fields by name and ignores unknown keys, an old backup still imports into a newer app (the missing field takes its default), which is exactly why the file has a version number. A sensible next course would be a second Catalogue course adding barcode scanning with a lookup, and the data-model changes that go with it.

Compared with the iOS Course

Personal Catalogue: iOS (Swift & SwiftUI) builds the same ten chapters. Side by side, the whole project:

CHAPTER	ANDROID (THIS COURSE)	IOS (SISTER COURSE)
1 Setup	Android Studio, Gradle, emulator	Xcode, Swift Package defaults, simulator
2 Data model	Room entities, an explicit cross-reference class	SwiftData `@Model` classes; the join is implicit
3 Persistence	DAOs, database class, repository, Hilt	`ModelContainer`; no injection library
4-5 List, detail	ViewModel + StateFlow, LazyColumn, Navigation Compose	`@Query`, `List`, `NavigationStack`
6-7 Form, tags	Hoisted state in a view model, chips with FlowRow	`Form`, bindings, a tag editor in SwiftUI
8 Search	SQL query with debounce and flatMapLatest	`.searchable` and a `#Predicate`
9 Polish	Dialogs, theme checks, TalkBack, snackbars	Alerts, automatic dark mode, VoiceOver labels

CHAPTER	ANDROID (THIS COURSE)	IOS (SISTER COURSE)
10 Backup, tests	Storage Access Framework, JUnit, in-memory Room	`fileExporter` / `fileImporter`, Swift Testing or XCTest

The honest summary: Android asks you to assemble more pieces yourself and gives you more control over where each one sits; iOS provides more of them already joined up. Neither is wrong, and having built both you can choose with your eyes open, which is the real goal of building one app twice.

What You Have Built

- A native Android app in Kotlin and Jetpack Compose, with Room persistence, entirely offline.
- A layered architecture (entities, DAOs, repository, view models, screens) in which each layer can be tested and replaced.
- Features: add, edit and delete with per-type forms; a managed tag system; live search and filters; the "do I already own this?" flow; a duplicate warning; light and dark themes; accessibility basics; explicit error states; and a portable JSON backup.
- Habits worth keeping: state hoisting, one source of truth, atomic transactions, never trusting an id you did not re-read, and testing the rules in order of cheapness.

Challenges

EXERCISE 1

Add the backup code and the Backup screen. Add six items, export, open the file in a text editor, then clear the app's storage and import. Record the message after the first import, then import the *same* file again and record that message. Edit one item's title inside the JSON file by hand and import that file: what happens, and why?

View solution

EXERCISE 2

Write and run the pure-function tests, then break the app on purpose: remove the "escape the backslash first" line from `likePattern`. Which test fails, and what does its message say? Put the line back. Then write one more test of your own for a rule from Chapters 2 to 9.

View solution

EXERCISE 3

Design (no need to build) the ISBN feature from the roadmap: list the data-model change, the Room migration, the new form field and label, the change to the backup format, and what an old backup file does when imported into the new app. Then say what a *new* backup does when imported into the *old* app.

View solution

CHAPTER 10 QUICK REFERENCE

- Backup file: JSON with a `version`, items stored with type and tags *by name*, no database ids; `ignoreUnknownKeys = true` for forward compatibility
- Import merges, skips items that match on title, creator, type and creation time, and is all-or-nothing in one `withTransaction`; decode the JSON before the transaction
- Storage Access Framework: `CreateDocument("application/json")` to save, `OpenDocument()` to open; the user picks the file; no storage permission needed
- `rememberLauncherForActivityResult(contract) { uri -> ... }`; `uri` is `null` if cancelled
- File work on `Dispatchers.IO`: `contentResolver.openOutputStream(uri, "wt")` / `openInputStream(uri)`
- Tests: pure functions (local, fast) → repository on an in-memory Room database (instrumented) → view model with a test main dispatcher
- Changing an installed database means raising its version and adding a `Migration`

NOT YET VERIFIED

Checked (two web searches on 2026-11-19): that the Storage Access Framework uses `ACTION_CREATE_DOCUMENT` and `ACTION_OPEN_DOCUMENT`, that the user selects the file and the app receives a URI permission for it; that `rememberLauncherForActivityResult` takes any `ActivityResultContract` and handles registering and unregistering; that Google's Room testing page

recommends an in-memory database with `runTest` and closing it after each test, and describes newer setups I did not follow.

Not checked, from general knowledge: `ContentResolver.openOutputStream(uri, "wt")` and `openInputStream` in this combination; `CreateDocument` and `OpenDocument` constructor and `launch` signatures in your activity-compose version; `kotlinx.serialization` configuration options and behaviour of the generated serializers for the default values; that a thrown exception inside `withTransaction` rolls everything back; `Icons.Filled.Settings`; the migration class shape for your Room version; every test, including the ViewModel test's use of `setMain` with `first { }`; the JUnit and AndroidX test dependency setup; ML Kit and CameraX as the barcode route; and all code, **none of which was compiled, and no test was run**.

SOURCES FOR THIS CHAPTER

Search results (2026-11-19): Android Developers, DocumentsProvider; ManagedActivityResultLauncher API reference; Compose and other libraries; Test and debug your database (Room); Migrate your Room database; Lifecycle release notes. Only titles and short summaries in the search results were used.

COURSE COMPLETE

That is the last chapter of *Personal Catalogue: Android (Kotlin & Compose)*. If you also build *Personal Catalogue: iOS (Swift & SwiftUI)*, compare your two finished apps against the table above, and note which parts felt natural on each platform: that judgement is the lasting product of this pair of courses.