

ANDROID COURSE

Android Development with Compose: Architecture & Data

11 Chapters

Philip Osztromok

Generated with Claude

ANDROID DEVELOPMENT WITH COMPOSE: ARCHITECTURE & DATA

Contents

Chapter 1	Architecture and UI State	2
Chapter 2	ViewModel and StateFlow	9
Chapter 3	Coroutines in Android	16
Chapter 4	Room Database	23
Chapter 5	The Repository Pattern	30
Chapter 6	Kotlin Serialization and JSON	39
Chapter 7	Networking with Retrofit	46
Chapter 8	DataStore and Preferences	54
Chapter 9	Manual Dependency Injection	61
Chapter 10	Hilt	68
Chapter 11	Capstone Notes App	76

Android Development with Compose: Architecture & Data

Course 5 · Chapter 1 · Organising an App

Android Development Fundamentals with Compose taught you to build screens. This course is about what sits *behind* them: where data lives, how it reaches the screen, and how the code stays understandable as the app grows. Before any new library, this chapter gives you the picture: layers, the way state and events travel, and the small class that describes what a screen is showing.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked in the docs (one web search, summaries only):** the recommended layers (UI and data, with an optional domain layer), unidirectional data flow, exposing a single `uiState` as a `StateFlow`, modelling it as a data class or a sealed class, repositories between the UI and data sources, and not sending one-off events from a ViewModel to the UI. **General knowledge, not checked:** the "no architecture" example, the folder layout, and the MVVM wording. **All code is unrun** (no Kotlin compiler on the machine used to write this). Nothing here was seen on your install. The names `NotesUiState`, `NotesScreen` and the `ComposeHelloTheme` in the previews are examples; use your own project's.

The Problem Architecture Solves

Here is a screen written the easy way, with everything inside the composable:

```
// Everything in one composable: the "no architecture" version
@Composable
fun NotesScreen() {
    var notes by remember { mutableStateOf(listOf<String>()) }
    var loading by remember { mutableStateOf(true) }

    LaunchedEffect(Unit) {
        // network call, database query and parsing all live here
        notes = loadNotesFromWhereverItLives()
        loading = false
    }
```

```
    if (loading) Text("Loading...") else notes.forEach { Text(it) }
}
```

(general knowledge) It works, until it grows. Typical troubles:

- **It cannot be tested** without drawing a screen, because the logic is welded to the UI.
- **It cannot be previewed** in Android Studio, because the preview would try to call the network.
- **State is lost on rotation** (Chapter 2 of Android Development Fundamentals with Compose), and every screen repeats the same loading code.
- **Several screens need the same data**, so the same fetching code is copied around.

Layers

(docs) Android's architecture guidance recommends **at least two layers**, plus an optional third:

LAYER	JOB	LIVES IN
UI layer	Displays application data and takes user input. In Compose: your composables, plus the state holder (a ViewModel) that prepares what they show.	`ui/`
Data layer	Holds the business logic and exposes app data. Repositories sit here and hide where data comes from (database, network, files).	`data/`
Domain layer (optional)	Reusable business rules ("use cases"). The docs say it is for reusing logic across ViewModels or simplifying a complex one, and mainly worth it for big apps.	`domain/`

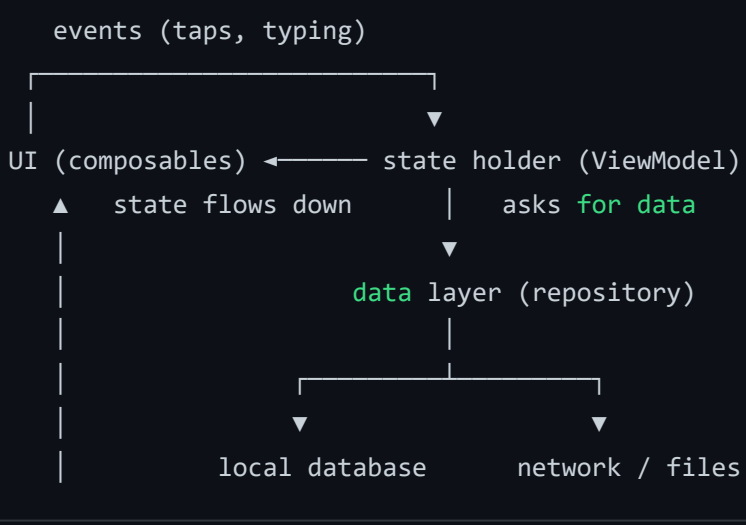
(docs) Two of the recommendations that matter most: repositories sit between the UI and the data sources, so UI components never touch a database or network client directly; and layers talk to each other with coroutines and flows (Chapter 3).

```
com.example.notes/
├── ui/
│   ├── notes/
│   │   ├── NotesScreen.kt        (composables)
│   │   ├── NotesUiState.kt       (the state class)
│   │   └── NotesViewModel.kt     (Chapter 2)
│   └── theme/
```

```
├── data/
│   ├── NotesRepository.kt        (Chapter 5)
│   ├── local/                    (Room: Chapter 4)
│   └── remote/                   (Retrofit: Chapter 7)
├── domain/                       (optional: use cases)
└── MainActivity.kt
```

Unidirectional Data Flow

(docs) **Unidirectional data flow** (UDF) means **state flows down** from a state holder to the UI, and **events flow up** from the UI. A tap becomes an event passed upward; the state holder updates the state; the UI redraws from the new state. It is the state hoisting you met in Android Development Fundamentals with Compose 4, applied to the whole screen.



The benefit: there is one place where the state changes, so a bug in what is shown has one place to look.

UI State: One Class for What the Screen Shows

(docs) The recommendation is to expose a **single** `uiState` property from the state holder, modelled as a **data class** holding the data, errors and a loading flag, or a **sealed class** when the states are mutually exclusive. Both shapes:

```
data class NotesUiState(
    val notes: List<String> = emptyList(),
```

```
    val isLoading: Boolean = false,
    val errorMessage: String? = null
)
```

```
sealed interface NotesUiState {
    data object Loading : NotesUiState
    data class Success(val notes: List<String>) : NotesUiState
    data class Error(val message: String) : NotesUiState
}
```

Both are called `NotesUiState` here, so use one at a time; the screen below uses the sealed interface.

	DATA CLASS (FLAGS AND FIELDS)	SEALED INTERFACE (ONE CASE AT A TIME)
Fits when	Several things can be true together (content showing while a refresh runs)	The screen is exactly one of loading, content or error
Impossible states	Possible (loading and error at once)	Not representable
Reading it in the UI	`if` on the fields	An exhaustive `when`

A Stateless Screen from a UI State

With a state class, the screen becomes a plain function from state to UI, with events as callbacks. That also fixes the preview problem: pass in any state you like.

```
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.layout.Column
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
```

```
@Composable
fun NotesScreen(state: NotesUiState, onRetry: () -> Unit) {
    when (state) {
        NotesUiState.Loading -> Text("Loading...")
        is NotesUiState.Success -> LazyColumn {
            items(state.notes) { note -> Text(note) }
        }
        is NotesUiState.Error -> Column {
            Text(state.message)
            Button(onClick = onRetry) { Text("Try again") }
        }
    }
}

@Preview(showBackground = true)
@Composable
fun NotesScreenSuccessPreview() {
    ComposeHelloTheme {
        NotesScreen(state = NotesUiState.Success(listOf("Buy milk", "Call Ada")), onRetry
    }
}

@Preview(showBackground = true)
@Composable
fun NotesScreenErrorPreview() {
    ComposeHelloTheme {
        NotesScreen(state = NotesUiState.Error("No connection"), onRetry = { })
    }
}
```

(docs) The same advice applies to events: the docs say not to send one-off events from the ViewModel to the UI, but to process the event in the ViewModel and show the result through a state update. Chapter 2 builds the ViewModel that supplies this state.

MVVM in One Paragraph

(general knowledge, and the structure the old course called MVVM) **Model** is the data layer; **View** is the composables; **ViewModel** is the state holder in between: it asks the data layer for data and turns it into UI state. The diagram above is MVVM with Compose, with the arrows labelled the way Compose likes them.

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Architecture & Data 8 teaches the repository pattern, clean architecture basics and the separation of concerns in a View-based setting. The ideas are the same; the difference here is that Compose makes the state-down, events-up rule natural, and `StateFlow` replaces `LiveData` as the default way to publish state.

Try It Yourself

EXERCISE 1

Take the "no architecture" `NotesScreen` above and list each responsibility it has (drawing, holding state, loading, parsing, and so on). Sort them into UI layer, data layer or "state holder", and say what the screen composable would be left with.

View solution

EXERCISE 2

Pick a screen of your own (a weather forecast, a recipe list, a to-do list). Write a UI state for it twice: once as a data class and once as a sealed interface. Say which you prefer for that screen and why.

View solution

EXERCISE 3

Build the stateless `NotesScreen` and write three `@Preview` functions, one each for loading, success and error. Check all three in the preview without running the app.

View solution

NOT YET VERIFIED

Checked in the docs (one web search, summaries only): at least two layers with an optional domain layer; repositories between the UI and data sources; coroutines and flows between layers; unidirectional data flow (state down, events up); a single `uiState` exposed as a `StateFlow`; data class or sealed class for UI state; no one-off events from the ViewModel to the UI.

General knowledge, not checked: the problems list for the no-architecture version, the folder layout, the MVVM summary, the data class against sealed interface table.

Not verified: every listing and solution (unrun); the imports; the exercises (not run by me). Nothing was checked on your install.

Sources: Guide to app architecture (developer.android.com), Recommendations for Android architecture (developer.android.com), UI layer (developer.android.com), UI State production (developer.android.com), Compose UI Architecture (developer.android.com).

COMING UP

Chapter 2 builds the state holder: a `ViewModel` that exposes a `StateFlow` of UI state, how a composable collects it, and why the state survives rotation.

Android Development with Compose: Architecture & Data

Course 5 · Chapter 2 · The State Holder

Chapter 1 described a screen as a function from UI state to pixels, with events going back up. This chapter builds the thing in the middle: a **ViewModel** that holds the state, changes it when events arrive, and publishes it as a `StateFlow` for Compose to collect.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked in the docs (one web search, summaries only):** the pattern of a private `MutableStateFlow` exposed as a read-only `StateFlow` with `asStateFlow()`; getting the ViewModel in a composable with `viewModel()` and reading it with `collectAsStateWithLifecycle()`; collection starting at the STARTED lifecycle state by default and stopping below it, adjustable with `minActiveState`; and ViewModels surviving configuration changes such as rotation. **General knowledge, not checked:** the Gradle artifact names, the exact import paths, the Snackbar code, the LiveData comparison, and the process-death note. **One discrepancy the docs did not explain:** the Lifecycle release notes say the `collectAsStateWithLifecycle()` extensions arrived in version 2.6.0, while the current API reference marks them "Added in 2.10.0", so this chapter does not name a version. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

What a ViewModel Is For

(docs) A `ViewModel` is the UI layer's state holder. It lives longer than any one composition: it **persists across configuration changes such as screen rotation**, so the new Activity instance gets the same state back instead of starting again. That is the fix for the "state lost on rotation" problem from Chapter 1.

	`remember` / `rememberSaveable`	`ViewModel`
Best for	Small, purely visual state (a text field draft, a scroll position)	Screen state and the logic that changes it
Survives rotation	`remember`: no. `rememberSaveable`: yes	Yes *(docs)*
Can run logic and call the data layer	Awkward	Yes (Chapter 3 adds coroutines)
Testable without a screen	No	Yes (it is a plain class)

(general knowledge) A ViewModel does not survive the whole process being killed by the system. If a value must survive that too, the ViewModel can use a `SavedStateHandle`; this course mostly relies on Room and DataStore (Chapters 4 and 8) for anything that must last.

Setup

```
// app/build.gradle.kts (dependencies block); versions are placeholders
implementation("androidx.lifecycle:lifecycle-viewmodel-compose:<version>")
implementation("androidx.lifecycle:lifecycle-runtime-compose:<version>")
```

(general knowledge) These are the two artifacts that provide `viewModel()` and `collectAsStateWithLifecycle()`. A new project from the current Android Studio template may already include some of them; check your version catalog before adding duplicates, and take current versions from the docs or the IDE's suggestions.

The UI State

This chapter uses the data-class form from Chapter 1, with a field for a one-off message:

```
data class NotesUiState(
    val notes: List<String> = emptyList(),
    val isLoading: Boolean = false,
    val userMessage: String? = null
)
```

The ViewModel

```
import androidx.lifecycle.ViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.update

class NotesViewModel : ViewModel() {
```

```
    // Private and writable inside the ViewModel only...
    private val _uiState = MutableStateFlow(NotesUiState(isLoading = true))

    // ...exposed to the UI as read-only
    val uiState: StateFlow<NotesUiState> = _uiState.asStateFlow()

    init {
        loadNotes()
    }

    private fun loadNotes() {
        // Placeholder data: Chapters 3 to 7 replace this with real loading
        _uiState.update { it.copy(notes = listOf("Buy milk", "Call Ada"), isLoading = fa
    }

    fun addNote(text: String) {
        if (text.isBlank()) {
            _uiState.update { it.copy(userMessage = "A note cannot be empty") }
            return
        }
        _uiState.update { it.copy(notes = it.notes + text) }
    }

    fun messageShown() {
        _uiState.update { it.copy(userMessage = null) }
    }
}
```

- **Private writable, public read-only.** *(docs)* The `_uiState` / `uiState` pair is the recommended shape: only the ViewModel can change the state, and the UI can only read it.
- `update` **replaces the whole state from the old one** *(general knowledge)*, so two quick changes cannot overwrite each other with stale copies the way `_uiState.value = ...` built from an earlier read can.

Chapter 1's diagram.

- **No** `Context`**, no composables, no Activity.** *(docs)* The recommendations say to keep ViewModels free of Activity, `Context` and Resources references.
- **The data is fake for now.** `loadNotes` sets placeholder notes; Chapters 3 to 7 replace it with real loading. The ViewModel builds nothing else itself, which Chapter 9 (Manual Dependency Injection) will tidy up.

Collecting the State in Compose

```
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.lifecycle.viewmodel.compose.viewModel

@Composable
fun NotesRoute(viewModel: NotesViewModel = viewModel()) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    NotesScreen(
        state = uiState,
        onAddNote = viewModel::addNote,
        onMessageShown = viewModel::messageShown
    )
}
```

(docs) `collectAsStateWithLifecycle()` turns the `StateFlow` into Compose `State`, and only collects while the lifecycle is at least STARTED by default, so nothing is collected while the app is in the background. `by` unwraps the state so you can write `uiState` instead of `uiState.value`.

Two composables, one job each. `NotesRoute` is the *stateful* part: it knows about the ViewModel. `NotesScreen` is *stateless*, takes only state and callbacks, and stays previewable exactly as in Chapter 1. Keep `viewModel()` at the top of a screen, and pass plain values down.

```
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.Button
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Scaffold
import androidx.compose.material3.SnackbarHost
import androidx.compose.material3.SnackbarHostState
import androidx.compose.material3.Text
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
```

```
@Composable
fun NotesScreen(
    state: NotesUiState,
    onAddNote: (String) -> Unit,
    onMessageShown: () -> Unit
) {
    val snackbarHostState = remember { SnackbarHostState() }

    // Show the message once, then tell the ViewModel it has been shown
    LaunchedEffect(state.userMessage) {
        state.userMessage?.let { message ->
            snackbarHostState.showSnackbar(message)
            onMessageShown()
        }
    }

    Scaffold(snackbarHost = { SnackbarHost(snackbarHostState) }) { padding ->
        Column(modifier = Modifier.padding(padding).padding(16.dp)) {
            var draft by rememberSaveable { mutableStateOf("") }
            Row {
                OutlinedTextField(value = draft, onValueChange = { draft = it }, label =
                Button(onClick = { onAddNote(draft); draft = "" }) { Text("Add") }
            }
            if (state.isLoading) {
                Text("Loading...")
            } else {
                LazyColumn { items(state.notes) { note -> Text(note) } }
            }
        }
    }
}
```

```
// MainActivity: setContent { ComposeHelloTheme { NotesRoute() } }
```

One-Off Messages Without One-Off Events

(docs) The docs advise against sending one-off events from the ViewModel to the UI: process the event in the ViewModel and show the result by updating state. That is what `userMessage` does. The ViewModel sets the message; the screen shows it once and calls `onMessageShown()`,

which sets it back to `null`. After a rotation the state still says what to show, so nothing is lost or shown twice by accident. *(general knowledge)* The exact Snackbar and `LaunchedEffect` handling above is one way to do it; the docs' own samples may differ in detail.

A Note on LiveData

(general knowledge) Older code publishes state with `LiveData`. It still works, and Compose can read it with `observeAsState()`, but it belongs to the View era. New code uses `StateFlow`, as the docs' recommendation of a `StateFlow` named `uiState` says.

```
// The old way (Android Development: Architecture & Data 2)
private val _notes = MutableLiveData<List<String>>()
val notes: LiveData<List<String>> = _notes
// In Compose you would call notes.observeAsState(); new code uses StateFlow instead
```

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Architecture & Data 2 covers the ViewModel lifecycle, LiveData, StateFlow and UI state patterns. The ViewModel here is the same idea. What changes is how the UI reads it: a composable collects with `collectAsStateWithLifecycle()`, and no `observe()` call, no `lifecycleOwner` and no view binding are needed.

Try It Yourself

EXERCISE 1

Build `NotesViewModel`, `NotesRoute` and `NotesScreen`, run the app, add a note, then rotate the emulator. Report whether the notes survived and what the Chapter 1 screen (with plain `remember`) would have done.

View solution

EXERCISE 2

Add a `removeNote(note: String)` function to the ViewModel and a button next to each note that calls it. Explain why the ViewModel exposes `StateFlow` and not `MutableStateFlow`.

View solution

EXERCISE 3

Put a counter in a `remember { mutableStateOf(0) }` in the screen and a second counter in the ViewModel's state. Increase both, rotate, and report which survived. Then say which one you would use for a text field draft and which for the list of notes.

View solution

NOT YET VERIFIED

Checked in the docs (one web search, summaries only): private `MutableStateFlow` exposed as `StateFlow` via `asStateFlow()`; `viewModel()` and `collectAsStateWithLifecycle()`; collection from STARTED, adjustable with `minActiveState`; ViewModels persisting across configuration changes; no Activity, Context or Resources in a ViewModel; no one-off events from the ViewModel to the UI.
General knowledge, not checked: the Gradle artifact names; the import paths; the `update` remark; the Snackbar and `LaunchedEffect` code; process death and `SavedStateHandle`; the LiveData comparison and `observeAsState()`.
Docs discrepancy, unresolved: `collectAsStateWithLifecycle()` is said to be added in 2.6.0 in the release notes but "Added in 2.10.0" in the API reference; no version is claimed here.
Not verified: every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.
Sources: ViewModel overview (developer.android.com), collectAsStateWithLifecycle API reference (developer.android.com), Recommendations for Android architecture (developer.android.com), Kotlin coroutines with lifecycle-aware components (developer.android.com), Lifecycle releases (developer.android.com).

COMING UP

Chapter 3 adds coroutines: how `viewModelScope` lets the ViewModel load data without freezing the screen, what the dispatchers do, and how a `Flow` from the data layer feeds this same state.

Android Development with Compose: Architecture & Data

Course 5 · Chapter 3 · Doing Work Off the Screen

Real data takes time to arrive: a database read, a network call. If the app waits on the main thread, the screen freezes. **Coroutines** let code pause without blocking, and give the ViewModel a safe way to start work, stop it, and turn a stream of data into UI state. The coroutine basics (`suspend`, `launch`, `delay`) come from the Kotlin courses; this chapter is about using them in an Android app.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked in the docs (one web search, summaries only): `viewModelScope` cancels its coroutines when the ViewModel is cleared and is hardcoded to `Dispatchers.Main`; inject dispatchers and replace Main in tests with `Dispatchers.setMain`; repository code should be main-safe; the data layer exposes suspend functions and Flows; build `uiState` with `stateIn` and `WhileSubscribed(5000)` for data that arrives as a stream; do not launch asynchronous work in a ViewModel's `init` or constructor; catch exceptions in coroutines; and for work of around five seconds or more use WorkManager. **The search did not cover `Dispatchers.IO` or `Dispatchers.Default` specifically**, so that table is general knowledge. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

A CORRECTION TO CHAPTER 2

The Chapter 2 ViewModel called `loadNotes()` from `init`. It was placeholder code with no real waiting, so it was harmless there, but *(docs)* the guidance is not to launch asynchronous operations in a ViewModel's `init` or constructor. This chapter shows the shape to use once loading is real: the state is built from the data layer's `Flow` with `stateIn`.

Why Not Just Wait?

```
// Blocking the main thread: the screen freezes while this runs
fun addNote(text: String) {
    Thread.sleep(3000)               // stands in for slow work
    _uiState.update { it.copy(notes = it.notes + text) }
}
```

(general knowledge) The main thread draws the screen and handles taps. Block it for a few seconds and the app stops responding (Android may show an "application not responding" dialog). The fix is not a faster sleep; it is to *not block*.

```
// A suspend function can pause without blocking the thread it runs on
suspend fun addNote(text: String) {
    delay(3000)                     // pauses this coroutine only
    // ...then continues here
}

// Starting a coroutine from the ViewModel
fun onAddClicked(text: String) {
    viewModelScope.launch {
        repository.addNote(text)
    }
}
```

A `suspend` function can pause at a point such as `delay` and let the thread do other work meanwhile. Only a coroutine (or another suspend function) can call it, which is why `viewModelScope.launch` is there.

viewModelScope

```
// What viewModelScope is, in outline
// - created with the ViewModel
// - cancelled automatically when the ViewModel is cleared
// - runs on Dispatchers.Main by default
```

- *(docs)* Coroutines launched in `viewModelScope` are **cancelled automatically when the ViewModel is cleared**, so work does not outlive the screen that wanted it.
- *(docs)* The scope is **hardcoded to** `Dispatchers.Main`. That is fine for starting work and updating state; heavy work must move to another dispatcher.
- *(docs)* A coroutine in this scope keeps running whether or not the UI is visible, until it finishes, is cancelled, or the ViewModel is cleared. For work of around five seconds or more that must finish regardless, the docs point to WorkManager (Android Development with Compose: Production & Publishing, planned).

Dispatchers and Main-Safety

DISPATCHER	USE IT FOR
`Dispatchers.Main`	The UI thread: updating state, light work. *(docs: the default for `viewModelScope`)*
`Dispatchers.IO`	Waiting on disk or network *(general knowledge)*
`Dispatchers.Default`	CPU-heavy work such as sorting or parsing a large list *(general knowledge)*

(docs) The advice is that the data layer's functions should be **main-safe**: safe to call from the main thread, because they switch dispatchers themselves with `withContext`. The caller does not need to know. That is how `FakeNotesRepository.addNote` below is written. The docs also say to **inject the dispatcher** rather than hard-code it, so a test can replace it:

```
// Injecting the dispatcher so a test can replace it
class FakeNotesRepository(
    private val ioDispatcher: CoroutineDispatcher = Dispatchers.IO
)
```

A Repository That Exposes Suspend Functions and a Flow

```
import kotlinx.coroutines.CoroutineDispatcher
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.delay
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.update
import kotlinx.coroutines.withContext

// A fake data source for now: Chapters 4 to 7 replace it with Room and Retrofit
class FakeNotesRepository(
    private val ioDispatcher: CoroutineDispatcher = Dispatchers.IO
) {
    private val _notes = MutableStateFlow(listOf("Buy milk", "Call Ada"))

    // A Flow: it emits the current list now and again after every change
    val notes: Flow<List<String>> = _notes.asStateFlow()
```

```
    // Main-safe: callers can use it from the main thread
    suspend fun addNote(text: String) = withContext(ioDispatcher) {
        delay(300)                  // pretend this is slow
        if (text == "fail") error("Could not save")
        _notes.update { it + text }
    }
}
```

- `notes` is a `Flow` *(docs: the data layer exposes suspend functions and Flows)*: the repository says "here is the list, and I will tell you when it changes".
- `addNote` is a **suspend function** that does its slow work on the injected IO dispatcher.
- It fails when the text is `"fail"`, so you can see the error path.

From Flow to UI State with stateIn

```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.CancellationException
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

class NotesViewModel(
    private val repository: FakeNotesRepository = FakeNotesRepository()
) : ViewModel() {

    private val message = MutableStateFlow<String?>(null)

    // Built from the data layer's Flow: no loading code in init
    val uiState: StateFlow<NotesUiState> =
        combine(repository.notes, message) { notes, userMessage ->
            NotesUiState(notes = notes, isLoading = false, userMessage = userMessage)
        }.stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5_000),
            initialValue = NotesUiState(isLoading = true)
        )
```

```
    fun addNote(text: String) {
        if (text.isBlank()) {
            message.value = "A note cannot be empty"
            return
        }
        viewModelScope.launch {
            try {
                repository.addNote(text)
            } catch (e: CancellationException) {
                throw e                          // never swallow cancellation
            } catch (e: Exception) {
                message.value = "Could not save the note"
            }
        }
    }

    fun messageShown() {
        message.value = null
    }
}
```

- `combine` *(general knowledge)* merges the repository's notes with the current message into one `NotesUiState` each time either changes.
- `stateIn(...)` *(docs)* turns that cold flow into a `StateFlow` that the UI can collect. The docs' recommended policy for data arriving as a stream is `SharingStarted.WhileSubscribed(5000)`: the upstream stays active for five seconds after the last collector goes away, which covers a rotation without restarting the work, but stops it if the user really leaves.
- `initialValue` is what the UI shows before the first emission, here the loading state.
- Nothing is launched in `init`: the work starts when something collects `uiState`.

The Chapter 2 composables are unchanged: `NotesRoute` still calls `collectAsStateWithLifecycle()`. *(general knowledge)* The default value on the constructor parameter is what lets the standard `viewModel()` create this class without a factory (a class whose parameters all have defaults gets a no-argument constructor on the JVM); Chapter 9 replaces that with proper dependency injection.

Exceptions and Cancellation

```
try {
    repository.addNote(text)
} catch (e: CancellationException) {
    throw e                        // let the cancellation pass through
} catch (e: Exception) {
    message.value = "Could not save the note"
}
```

(docs) Exceptions in coroutines should be caught, because an unhandled one can crash the app, and the docs' own sample rethrows `CancellationException` before catching general exceptions. *(general knowledge)* The reason: cancellation is delivered as a `CancellationException`; if you catch it as an ordinary error and carry on, the coroutine does not actually stop. Here a failed save becomes a message in the UI state instead of a crash.

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Architecture & Data 6 covers `viewModelScope`, `lifecycleScope` and the IO and Main dispatchers. In a Compose app you rarely need `lifecycleScope`: the ViewModel starts the work, and `collectAsStateWithLifecycle()` (Chapter 2) handles the UI side of the lifecycle.

Try It Yourself

EXERCISE 1

Replace the Chapter 2 ViewModel with the `stateIn` version and add `FakeNotesRepository`. Run it, add a note, and report whether the screen stays responsive during the 300 ms delay and whether you see the loading state at start-up.

View solution

EXERCISE 2

Add a note with the text `fail` and confirm the Snackbar from Chapter 2 shows "Could not save the note". Then remove the `CancellationException` branch and explain, in two sentences, what could go wrong.

View solution

EXERCISE 3

Explain what `WhileSubscribed(5_000)` does when you rotate the device, and what it does when you press Home and wait a minute. Then explain why the docs advise against loading from `init`.

View solution

NOT YET VERIFIED

Checked in the docs (one web search, summaries only): `viewModelScope` cancellation on clear; hardcoded to Main; inject dispatchers and `Dispatchers.setMain` in tests; main-safe repositories; suspend functions and Flows from the data layer; `stateIn` with `WhileSubscribed(5000)`; no asynchronous work in `init`; catching coroutine exceptions and rethrowing `CancellationException` in a sample; WorkManager for work of about five seconds or more.

General knowledge, not checked: the IO and Default dispatcher descriptions; the main-thread freezing explanation; `combine`; why cancellation must not be swallowed; the default-parameter constructor remark; the import paths.

Not verified: every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.

Sources: Best practices for coroutines in Android (developer.android.com), Kotlin coroutines with lifecycle-aware components (developer.android.com), UI State production (developer.android.com), Recommendations for Android architecture (developer.android.com).

COMING UP

Chapter 4 replaces the fake repository's list with a real Room database: entities, a DAO that returns a `Flow`, and the build setup it needs.

Android Development with Compose: Architecture & Data

Course 5 · Chapter 4 · A Real Database

Until now the notes lived in a fake list that vanished when the app closed. **Room** is Android's library for keeping data in a local SQLite database without writing the plumbing by hand. You describe a table as a class (an **entity**), the queries as an interface (a **DAO**), and the database as a class that ties them together. Room generates the rest.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked in the docs (two web searches, summaries only): Room now has two lines, Room 2.x (package `androidx.room`) and Room 3.0 (package `androidx.room3`, artifacts `room3-runtime` and `room3-compiler`); Room 3.0 only supports KSP (no KAPT or Java annotation processors) and you can move to KSP while still on 2.x; DAO methods that return a `Flow` need not be `suspend`, other DAO functions must be; coroutines and Flow are built into the 2.x runtime so no extra artifact is needed; in 3.0 migrations and callbacks use `SQLiteConnection` instead of `SupportSQLiteDatabase`; and the docs recommend moving first to Room 2.8 with KSP and coroutines, then to 3.0. **Not checked:** the KSP plugin id and version rules, the code below (written in the long-standing Room 2.x form), and anything about the 3.0 database-builder call. **I could not confirm which Room version is the current stable one** (the release page showed 3.0.3 while another docs page used a 3.1.0 alpha as an example), so every version here is a placeholder. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

Which Room? 2.x or 3.0

	ROOM 2.X	ROOM 3.0 *(DOCS)*
Package	`androidx.room`	`androidx.room3`
Artifacts	`room-runtime`, `room-compiler`	`room3-runtime`, `room3-compiler`
Code generation	KSP (KAPT also worked in older versions)	KSP only
DAO functions	Suspend for one-shot calls, Flow for observing	Same rule: suspend unless the return type is observable
Migrations	`SupportSQLiteDatabase`	`SQLiteConnection`

	ROOM 2.X	ROOM 3.0 *(DOCS)*
Other reactive types (LiveData, RxJava, Paging)	Supported	Need registered `@DaoReturnTypeConverters` from their own artifacts

How this chapter handles it. The code below uses the long-standing Room 2.x form, with `androidx.room` imports. *(docs)* Moving to 3.0 is described as mostly a rename of artifacts and imports plus the migration changes above, and the recommended route is to get onto Room 2.8 with KSP and coroutines first. **Before you start a real project, read the current Room release notes and the migration guide and pick one line**; if you pick 3.0, change the imports to `androidx.room3` and check the database builder against the migration guide, because that part was not checked here.

Setup

```
// Room 2.x: module build.gradle.kts (versions are placeholders)
plugins {
    id("com.google.devtools.ksp") version "<ksp-version>"
}

dependencies {
    implementation("androidx.room:room-runtime:<room-version>")
    ksp("androidx.room:room-compiler:<room-version>")
}
```

```
// Room 3.0: the artifacts and the package are renamed (check the current version)
dependencies {
    implementation("androidx.room3:room3-runtime:<room-version>")
    ksp("androidx.room3:room3-compiler:<room-version>")
}
```

- *(docs)* The compiler dependency goes through `ksp`, not `kapt`. *(docs)* No separate coroutines artifact is needed for Room 2.x.
- *(general knowledge)* The KSP plugin id `com.google.devtools.ksp` and the need to match its version to your Kotlin version. The generated project uses AGP 9.x and Kotlin 2.2 (Android

Studio 2: The New UI 4), so take the matching KSP version from the KSP release notes; with a version catalog you would write the plugin and libraries there instead.

An Entity: One Row, One Class

```
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "notes")
data class NoteEntity(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    val text: String,
    val createdAt: Long = System.currentTimeMillis()
)
```

- `@Entity` makes the class a table; `@PrimaryKey(autoGenerate = true)` lets the database number the rows *(general knowledge)*.
- It is a **storage** class. The UI will use a separate model, mapped from this one in Chapter 5, so a change to the table does not ripple through the screens.

A DAO: The Questions You Can Ask

```
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Query
import kotlinx.coroutines.flow.Flow

@Dao
interface NoteDao {

    // A Flow: emits the list now and again whenever the table changes. Not suspend.
    @Query("SELECT * FROM notes ORDER BY createdAt DESC")
    fun observeNotes(): Flow<List<NoteEntity>>

    // One-shot writes are suspend functions
    @Insert
    suspend fun insert(note: NoteEntity)
```

```
    @Delete
    suspend fun delete(note: NoteEntity)

    @Query("DELETE FROM notes")
    suspend fun clearAll()
}
```

(docs) A `Flow`-returning query emits again whenever the data changes, and does not need `suspend`. Writes are `suspend` functions, which Room runs off the main thread for you *(general knowledge)*, so they are main-safe in the sense of Chapter 3. Room checks the SQL in each `@Query` when it compiles, so a typo in a column name fails the build rather than the app *(general knowledge)*.

The Database Class

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

@Database(entities = [NoteEntity::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun noteDao(): NoteDao
}
```

```
import android.app.Application
import androidx.room.Room

class NotesApp : Application() {
    // One database for the whole app
    val database: AppDatabase by lazy {
        Room.databaseBuilder(
            applicationContext,
            AppDatabase::class.java,
            "notes.db"
        ).build()
    }
}
```

```
// AndroidManifest.xml: <application android:name=".NotesApp" ... >
```

(general knowledge) `version` is the schema version. Change an entity and you must raise it and provide a migration, or the existing database on a user's phone cannot be opened. `exportSchema = false` keeps this chapter simple; real apps export schemas to track changes, which the docs cover under migrations. Build one database for the whole app, not one per screen.

A Search Query With a Parameter

```
@Query("SELECT * FROM notes WHERE text LIKE '%' || :term || '%' ORDER BY createdAt DESC")
fun search(term: String): Flow<List<NoteEntity>>
```

(general knowledge) `:term` refers to the function's parameter; Room binds it safely, so user text is never pasted into the SQL.

Testing a DAO

(general knowledge) The usual test builds an **in-memory** database that vanishes when closed, and checks a query against it. Android Development with Compose: Production & Publishing (planned) covers testing properly; here is the shape, which also needs the AndroidX test and coroutines-test libraries:

```
// app/src/androidTest/.../NoteDaoTest.kt
class NoteDaoTest {
    private lateinit var db: AppDatabase
    private lateinit var dao: NoteDao

    @Before
    fun createDb() {
        val context = ApplicationProvider.getApplicationContext<Context>()
        db = Room.inMemoryDatabaseBuilder(context, AppDatabase::class.java).build()
        dao = db.noteDao()
    }
```

```
    @After
    fun closeDb() = db.close()

    @Test
    fun insertedNoteIsObserved() = runTest {
        dao.insert(NoteEntity(text = "Buy milk"))
        val notes = dao.observeNotes().first()
        assertEquals(listOf("Buy milk"), notes.map { it.text })
    }
}
```

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Architecture & Data 3 teaches entities, DAOs, the database class and coroutines with Room, with the annotation processor set up through `kapt` in the older edition. The idea is the same. Now the compiler runs through `ksp`, a `Flow` from the DAO replaces `LiveData`, and Room 3.0 is a real choice you have to make.

Try It Yourself

EXERCISE 1

Add the dependencies, `NoteEntity`, `NoteDao` and `AppDatabase` to a project and build it. Report the Room and KSP versions you used, where you found them, and any build error. A Room compile error often means a typo in an annotation or query.

View solution

EXERCISE 2

Add the `search` query to the DAO. Then say what must change if you add a `pinned: Boolean` column to `NoteEntity` after the app has already been installed on a phone.

View solution

EXERCISE 3

Write the in-memory DAO test from the chapter, add the test dependencies it needs, and run it. Report whether it passed, and what you had to add that the chapter did not mention.

View solution

NOT YET VERIFIED

Checked in the docs (two web searches, summaries only): the Room 2.x and Room 3.0 lines and their package and artifact names; KSP only in 3.0 and usable from 2.x; the DAO suspend and Flow rule; no extra coroutines artifact in 2.x; `SQLiteConnection` migrations in 3.0; the recommended two-step path through Room 2.8; the existence of `@DaoReturnTypeConverters` for other reactive types.
General knowledge, not checked: the KSP plugin id and version matching; `@Entity`, `@PrimaryKey`, `@Insert`, `@Delete` and the database builder as shown; that Room checks SQL at compile time and runs suspend writes off the main thread; `exportSchema`; the in-memory test.
Not verified: which Room version is the current stable one (placeholders are used); the Room 3.0 database-builder call; every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.
Sources: Room 3.0 releases (developer.android.com), Migrate from Room 2.x to Room 3.0 (developer.android.com), Access data using Room DAOs (developer.android.com), Write asynchronous DAO queries (developer.android.com), Persist data with Room codelab (developer.android.com), Room 3.0: Modernizing the Room (Android Developers Blog).

COMING UP

Chapter 5 puts a repository in front of the DAO: one source of truth for the notes, mapping `NoteEntity` to the model the UI uses, and replacing Chapter 3's fake repository.

Android Development with Compose: Architecture & Data

Course 5 · Chapter 5 · One Door to the Data

Chapter 3 gave the ViewModel a fake repository and Chapter 4 gave you a real DAO. The **repository** joins them: a class the rest of the app talks to for notes, that decides where the notes come from. Today it hides Room; in Chapter 7 it will also hide a network call, and no screen will need to change.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked in the docs (one web search, summaries only):** repositories are recommended even when they wrap one data source; the UI is driven from data models that respect the single source of truth; a local database is the recommended source of truth when offline-first support matters; a class exposing a `Flow` follows a `getXxxStream` naming pattern (for example `getAuthorStream()`); a repository maps internal data models to external ones inside a Flow `map` (shown in a codelab the docs now mark deprecated); for complex apps create a model per layer; and state holders should never have a data source as a direct dependency. **General knowledge, not checked:** the interface and fake design, the factory code, the unit test and the layer table. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install. The code assumes the Room 2.x form from Chapter 4.

What a Repository Is For

- **One door.** *(docs)* Data sources stay hidden behind the repository; ViewModels and use cases never depend on a data source directly.
- **A single source of truth.** *(docs)* The UI shows data from one place. With a local database that is the recommended source of truth for offline-first support, the pattern is: the network (Chapter 7) writes into the database, and the screen only ever reads from the database.
- **Even for one data source.** *(docs)* The recommendation to use repositories applies even when one wraps a single data source, as ours does today.

A Model for the UI, Separate From the Entity

```
// The model the UI and the ViewModel use: plain data, no Room annotations
data class Note(
    val id: Long,
    val text: String,
    val createdAt: Long
)

// Mapping between the storage class (Chapter 4) and the model
fun NoteEntity.toModel() = Note(id = id, text = text, createdAt = createdAt)

fun Note.toEntity() = NoteEntity(id = id, text = text, createdAt = createdAt)
```

(docs) For complex apps, the guidance is a model per layer. *(general knowledge)* Here the reason is simple: `NoteEntity` is shaped by the database. If you add a column or rename a table, only the entity and the mapping change; the screens keep using `Note`. For a tiny app the two classes look identical, and that is fine: the habit is worth having before the app grows.

The Repository: an Interface and an Implementation

```
import kotlinx.coroutines.flow.Flow

interface NotesRepository {
    // Naming convention for a function that returns a Flow: getXxxStream
    fun getNotesStream(): Flow<List<Note>>

    suspend fun addNote(text: String)

    suspend fun deleteNote(note: Note)
}
```

```
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.map

class DefaultNotesRepository(
    private val noteDao: NoteDao
) : NotesRepository {

    override fun getNotesStream(): Flow<List<Note>> =
        noteDao.observeNotes().map { entities -> entities.map { it.toModel() } }
```

```
    override suspend fun addNote(text: String) {
        noteDao.insert(NoteEntity(text = text))
    }

    override suspend fun deleteNote(note: Note) {
        noteDao.delete(note.toEntity())
    }
}
```

- **The interface** says what the app can do with notes, in app terms (`Note`), with no mention of Room. *(docs)* The function returning a Flow is named `getNotesStream()`.
- **The mapping happens inside a Flow** `map` *(docs, in the codelab's example)*: entities go in, `Note`s come out.
- **No** `withContext` **here.** *(general knowledge)* Room's suspend writes run off the main thread themselves, so the repository is already main-safe in the Chapter 3 sense. If you add slow work of your own, switch dispatchers as in Chapter 3.

A Fake for Previews and Tests

```
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.update

// For previews and tests: same interface, no database
class FakeNotesRepository : NotesRepository {
    private val notes = MutableStateFlow(listOf(Note(1, "Buy milk", 0L)))

    override fun getNotesStream(): Flow<List<Note>> = notes

    override suspend fun addNote(text: String) {
        notes.update { it + Note(it.size + 1L, text, 0L) }
    }

    override suspend fun deleteNote(note: Note) {
        notes.update { it - note }
    }
}
```

Because the ViewModel will depend on the *interface*, a test can hand it this fake: no database, no emulator. This is the payoff of the interface. It replaces Chapter 3's `FakeNotesRepository`, which had no interface.

Updating the ViewModel and the State

```
data class NotesUiState(
    val notes: List<Note> = emptyList(),      // was List<String>
    val isLoading: Boolean = false,
    val userMessage: String? = null
)
```

```
class NotesViewModel(
    private val repository: NotesRepository      // the interface, not the Room class
) : ViewModel() {

    private val message = MutableStateFlow<String?>(null)

    val uiState: StateFlow<NotesUiState> =
        combine(repository.getNotesStream(), message) { notes, userMessage ->
            NotesUiState(notes = notes, isLoading = false, userMessage = userMessage)
        }.stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5_000),
            initialValue = NotesUiState(isLoading = true)
        )

    fun addNote(text: String) {
        if (text.isBlank()) {
            message.value = "A note cannot be empty"
            return
        }
        viewModelScope.launch {
            try {
                repository.addNote(text)
            } catch (e: CancellationException) {
                throw e
            } catch (e: Exception) {
                message.value = "Could not save the note"
            }
```

```
        }
    }

    fun deleteNote(note: Note) {
        viewModelScope.launch { repository.deleteNote(note) }
    }

    fun messageShown() {
        message.value = null
    }

    companion object {
        // A small hand-written factory: Chapter 9 explains and tidies this
        val Factory: ViewModelProvider.Factory = viewModelFactory {
            initializer {
                val app = this[ViewModelProvider.AndroidViewModelFactory.APPLICATION_KEY
                NotesViewModel(app.notesRepository)
            }
        }
    }
}
```

The ViewModel no longer builds its own repository. It **asks for one** in its constructor, which is what makes it testable. But the standard `viewModel()` can no longer create it, because the constructor needs an argument: so the companion object has a small `Factory` that reads the `NotesApp` and gives the ViewModel the app's repository. *(general knowledge)* That factory shape is the one the official codelab on repositories and manual dependency injection uses. Chapter 9 (Manual Dependency Injection) explains it properly and tidies it.

```
class NotesApp : Application() {
    val database: AppDatabase by lazy {
        Room.databaseBuilder(applicationContext, AppDatabase::class.java, "notes.db").bu
    }

    // The one place that builds the repository
    val notesRepository: NotesRepository by lazy {
        DefaultNotesRepository(database.noteDao())
    }
}
```

```
@Composable
fun NotesRoute(viewModel: NotesViewModel = viewModel(factory = NotesViewModel.Factory)) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    NotesScreen(
        state = uiState,
        onAddNote = viewModel::addNote,
        onDeleteNote = viewModel::deleteNote,
        onMessageShown = viewModel::messageShown
    )
}

// In the list, the screen now shows note.text:  items(state.notes) { note -> Text(note.
```

(general knowledge) The factory code also needs imports for `viewModelFactory`, `initializer` and `ViewModelProvider`; take their exact paths from the IDE's suggestions.

Who May Know What

CLASS	MAY USE	MUST NOT USE
Composables (`NotesScreen`)	`NotesUiState`, `Note`	Not the ViewModel itself, not the DAO
`NotesViewModel`	`NotesRepository` (the interface), `Note`	Not `NoteDao`, not `NoteEntity`
`DefaultNotesRepository`	`NoteDao`, `NoteEntity`, `Note`	Not composables, not the ViewModel
`NoteDao`, `AppDatabase`	`NoteEntity`	Not `Note`

(docs) The point of the last rule in the docs' summary: a state holder (the ViewModel) should never have a data source as a direct dependency. If a change to the database forces a change in a composable, a boundary has leaked.

Testing the ViewModel With the Fake

```kotlin
// app/src/test/.../NotesViewModelTest.kt  (a plain JVM test: no emulator)
@OptIn(ExperimentalCoroutinesApi::class)
class NotesViewModelTest {

    @Before
    fun setUp() = Dispatchers.setMain(UnconfinedTestDispatcher())

    @After
    fun tearDown() = Dispatchers.resetMain()

    @Test
    fun emptyNoteShowsMessage() = runTest {
        val viewModel = NotesViewModel(FakeNotesRepository())
        viewModel.addNote("   ")
        assertEquals("A note cannot be empty", viewModel.uiState.value.userMessage)
    }
}
```

(docs) The coroutines guidance says to replace Main in tests with `Dispatchers.setMain` and a test dispatcher, as above. *(general knowledge)* The test needs the `kotlinx-coroutines-test` library and a few imports; the details are covered when testing gets its own treatment in Android Development with Compose: Production & Publishing (planned).

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Architecture & Data 8 teaches the repository pattern, clean architecture basics and separating concerns, mostly with LiveData. The structure is the same. What is new here is the `getXxxStream` naming, `Flow` instead of `LiveData` through every layer, and a separate `Note` model between the entity and the screen.

Try It Yourself

EXERCISE 1

Add `Note`, the mapping functions, `NotesRepository` and `DefaultNotesRepository`; update the ViewModel, the factory, `NotesApp` and the screen. Run it, add three notes, close the app completely and reopen it. Report whether the notes were still there.

View solution

EXERCISE 2

Add `FakeNotesRepository` and write the ViewModel unit test. Add a second test that adds a real note and checks the list. Report the test dependencies you needed.

View solution

EXERCISE 3

Add a delete button to each note, wired all the way from the screen to the DAO. List every file you changed, and say whether any file that should not have changed did.

View solution

NOT YET VERIFIED

Checked in the docs (one web search, summaries only): repositories even for one data source; the single source of truth; a local database as the recommended source of truth for offline-first; `getXxxStream` naming; mapping inside a Flow `map` (a codelab now marked deprecated); a model per layer for complex apps; no data source as a direct dependency of a state holder; `Dispatchers.setMain` in tests (from the Chapter 3 search).
General knowledge, not checked: the interface and fake design; the factory code and its imports (the shape follows the official repository and manual-DI codelab, which was not read); Room suspend writes being main-safe; the unit test and its dependencies; the layer table.
Not verified: every listing and solution (unrun); the exercises (not run by me); the Room 3.0 form of the DAO and entity imports. Nothing was checked on your install.
Sources: Data layer (developer.android.com), Recommendations for Android architecture (developer.android.com), Build an offline-first app (developer.android.com), Building a Data Layer codelab (developer.android.com, deprecated), Add repository and Manual DI codelab (developer.android.com).

COMING UP

Chapter 6 is Kotlin Serialization and JSON: what JSON is, how to turn it into Kotlin classes, and how to cope with bad data, ready for the network call in Chapter 7.

Android Development with Compose: Architecture & Data

Course 5 · Chapter 6 · Turning Text Into Objects

Servers speak **JSON**: plain text that describes data. Your app speaks Kotlin classes. **kotlinx.serialization** is Kotlin's own library for converting between the two. This chapter is deliberately about the library on its own, with no network involved, so that when Chapter 7 adds Retrofit the only new thing is the request.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked in official Kotlin docs (one search limited to kotlinlang.org and GitHub, summaries only):** strict decoding is the default, so an unknown key is an error; `ignoreUnknownKeys = true` on a `Json` instance skips unknown keys; a `@JsonIgnoreUnknownKeys` annotation exists for one class, is experimental and needs an opt-in; `decodeFromString` decodes a JSON string using the deserializer for the type; and decoding failures come as a `JsonDecodingException`, part of a hierarchy under `SerializationException`. **Checked only in unofficial blog posts and Q&A (an earlier, unrestricted search):** `@SerialName` for keys that differ, defaults making a key optional, and that Retrofit needs a separate converter. **Not found in any source:** the Gradle plugin id and setup, `encodeToString`, `prettyPrint`, catching `IllegalArgumentException`, and the Gson and Moshi comparison. Those are general knowledge. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

What JSON Looks Like

```
{
  "userId": 1,
  "id": 7,
  "title": "Buy milk",
  "body": "Semi-skimmed, two litres"
}
```

(general knowledge) Curly braces hold an object, square brackets a list, and keys are always in quotes. Values are strings, numbers, `true` or `false`, `null`, objects or lists. This sample is in the shape of the free test service the next chapter uses; its exact fields get re-checked there.

Setup

```
// Module build.gradle.kts (versions are placeholders)
plugins {
    id("org.jetbrains.kotlin.plugin.serialization") version "<kotlin-version>"
}

dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:<serialization-vers
}
```

(general knowledge, not found in the searches) Serialization needs two things: a compiler plugin that generates the conversion code for classes you mark, and the runtime library with the JSON format. The plugin version follows your Kotlin version, so take both from the official setup page for the Kotlin version your project uses (Chapter 4 noted the generated project uses AGP 9.x and Kotlin 2.2).

A Serializable Class (a DTO)

```
import kotlinx.serialization.SerialName
import kotlinx.serialization.Serializable

@Serializable
data class RemoteNoteDto(
    val id: Long,
    val title: String,
    val body: String = "",                      // optional: has a default
    @SerialName("userId") val authorId: Long? = null   // JSON key differs from the Kotl
)
```

- `@Serializable` tells the plugin to generate the conversion for this class.
- **Defaults make a key optional.** *(blog sources)* `body` has a default, so JSON without a `body` key still decodes. A property with no default is required.
- `@SerialName` *(blog sources)* maps a JSON key to a Kotlin property with a different name; common for snake_case keys.

- **"DTO"** means *data transfer object (general knowledge)*: a class shaped like the network data, kept separate from `NoteEntity` and `Note` for the same reason as in Chapter 5.

The Json Instance, Decoding and Encoding

```
import kotlinx.serialization.json.Json

// One shared instance, configured once
val json = Json {
    ignoreUnknownKeys = true
    prettyPrint = true            // only affects encoding
}
```

```
val note: RemoteNoteDto = json.decodeFromString<RemoteNoteDto>(text)

val notes: List<RemoteNoteDto> = json.decodeFromString<List<RemoteNoteDto>>(listText)

val backToText: String = json.encodeToString(note)
```

(docs) `decodeFromString` uses the deserializer for the type you ask for. *(general knowledge)* `encodeToString` goes the other way. **Unknown keys:** the sample has a `userId` key; because the DTO maps it, it works. A key the DTO does not mention is an error by default; *(docs)* `ignoreUnknownKeys = true` skips it. That is the setting you almost always want for a server you do not control, because servers add fields.

```
// Without ignoreUnknownKeys, Json is strict: an unknown key is an error (docs).
// A per-class alternative exists but is experimental and needs an opt-in:
@OptIn(ExperimentalSerializationApi::class)
@JsonIgnoreUnknownKeys
@Serializable
data class LenientDto(val id: Long)
```

When the Data Is Bad

A missing required key, a string where a number should be, or text that is not JSON at all will fail. *(docs)* The failure comes as a `JsonDecodingException`, a kind of `SerializationException`. Catch it where you decode, and turn it into a result the rest of the app can handle:

```
import kotlinx.serialization.SerializationException

fun parseNotes(text: String): Result<List<RemoteNoteDto>> =
    try {
        Result.success(json.decodeFromString<List<RemoteNoteDto>>(text))
    } catch (e: SerializationException) {
        Result.failure(e)             // bad or unexpected JSON
    } catch (e: IllegalArgumentException) {
        Result.failure(e)             // a value the type could not accept
    }
```

(general knowledge) A value the type cannot accept can surface as `IllegalArgumentException` instead, which is why both are caught here. This matches Chapter 3's advice to catch exceptions in coroutines rather than let the app crash.

```
// The DTO is the network shape; map it to the storage class (Chapter 5's rule again)
fun RemoteNoteDto.toEntity() = NoteEntity(id = id, text = title)
```

A Test You Can Run Without a Phone

```
// app/src/test/.../RemoteNoteDtoTest.kt  (a plain JVM test)
class RemoteNoteDtoTest {

    @Test
    fun unknownKeysAreIgnored() {
        val text = "{\"id\":1,\"title\":\"Hi\",\"extra\":true}"
        val note = json.decodeFromString<RemoteNoteDto>(text)
        assertEquals("Hi", note.title)
        assertEquals("", note.body)                // the default was used
    }

    @Test(expected = SerializationException::class)
    fun missingRequiredFieldFails() {
        json.decodeFromString<RemoteNoteDto>("{\"id\":1}")
```

```
    }
}
```

(general knowledge) Parsing is plain Kotlin, so this runs as an ordinary unit test on your computer. It is the quickest way to try the library, and to see the error messages.

Gson, Moshi or kotlinx.serialization?

	GSON	MOSHI	KOTLINX.SERIALIZATION
Written for	Java	Java and Kotlin	Kotlin only
How it works	Reflection	Reflection or generated code	A compiler plugin generates the code
Knows Kotlin defaults and non-null types	Not reliably	With its Kotlin support	Yes: defaults make keys optional
Unknown keys	Ignored	Ignored	An error unless `ignoreUnknownKeys` *(docs)*

The Gson and Moshi columns, and the "how it works" row, are *(general knowledge, not checked)* and are a summary of common experience, not a benchmark. All three are used in real apps. The older course used Gson or Moshi; this course uses kotlinx.serialization because it is Kotlin-first and its strictness makes bad data visible instead of silent.

COMPARED WITH THE CLASSIC UI COURSE

This chapter has no counterpart in Android Development: Architecture & Data. That course covered JSON inside its Retrofit chapter (Android Development: Architecture & Data 4), using Gson or Moshi. Splitting it out here means Chapter 7 can concentrate on the network call.

Try It Yourself

EXERCISE 1

Add the plugin and library, create `RemoteNoteDto` and the `json` instance, and write the unit test from the chapter. Report the Kotlin, plugin and library versions you used, and whether the test passed.

View solution

EXERCISE 2

Break the decoding on purpose three ways: remove `ignoreUnknownKeys`, drop a required key, and put text where a number should be. Write down the exception type and message each time, and which one you would show a user.

View solution

EXERCISE 3

Add a `createdAt` property mapped from the JSON key `created_at` with `@SerialName`, make it nullable, and decode JSON with and without it. Explain when you would choose "nullable" and when "a default value".

View solution

NOT YET VERIFIED

Checked in official Kotlin docs (summaries only): strict by default; `ignoreUnknownKeys`; the experimental `@JsonIgnoreUnknownKeys` with an opt-in; `decodeFromString`; `JsonDecodingException` under `SerializationException`.

Unofficial sources only: `@SerialName`; defaults making keys optional; Retrofit needing a separate converter (Chapter 7 checks this).

General knowledge, not found in any search: the Gradle plugin id and setup; `encodeToString` and `prettyPrint`; catching `IllegalArgumentException`; the Gson, Moshi and kotlinx comparison; what JSON looks like; the sample's resemblance to the test service.

Not verified: every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.

Sources: JSON serialization overview (kotlinlang.org), Customize the Json instance (kotlinlang.org), ignoreUnknownKeys API reference (kotlinlang.org), JsonIgnoreUnknownKeys API reference (kotlinlang.org), Introduction to using Kotlin Serialization (ProAndroidDev, blog).

COMING UP

Chapter 7 makes the network call with Retrofit, uses this chapter's `Json` through a converter, and shows loading, success and error states, with the repository writing what it fetches into Room.

Android Development with Compose: Architecture & Data

Course 5 · Chapter 7 · Fetching From a Server

Now the notes can come from the internet. **Retrofit** turns an interface into HTTP calls; the Chapter 6 `Json` turns the reply into Kotlin objects; and following Chapter 5, the repository writes what it fetches into Room, so the screen keeps reading from one place and still works offline.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked in the docs (two searches, summaries only): a local database is the recommended source of truth for offline-first, and the repository fetches from the network and persists to the local source; Retrofit and Room suspend functions are main-safe (from a codelab the docs now mark deprecated); cleartext HTTP is off by default from Android 9 and is allowed per domain through a network security configuration, with the manifest flag being phased out. **Checked only in repositories and blog posts:** the converter is attached with `asConverterFactory` and a media type, and takes a configured `Json`. **Important:** the community converter by Jake Wharton is **deprecated**, and its repository says the converter moved into Retrofit as a first-party converter. **I did not confirm the first-party Gradle coordinates, package or function name**, so the code below uses the names I expect and says so; check Retrofit's own documentation first. **General knowledge, not found in any search:** the INTERNET permission, the Retrofit annotations, `HttpException`, the unique-index design and the test service's shape. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

Setup

```
<!-- AndroidManifest.xml, outside the <application> element -->
<uses-permission android:name="android.permission.INTERNET" />
```

```
// Module build.gradle.kts (versions are placeholders; the converter coordinates are to
dependencies {
    implementation("com.squareup.retrofit2:retrofit:<retrofit-version>")
    implementation("com.squareup.retrofit2:converter-kotlinx-serialization:<retrofit-ver
}
```

- *(general knowledge)* Without the `INTERNET` permission the app cannot reach the network at all; the search found no current page on it, so this is not docs-checked.
- **The converter.** Search results say the community converter is deprecated and has moved into Retrofit. The coordinates above are my best expectation of the official one. **Confirm them in Retrofit's documentation or release notes before use**; the old community artifact (last release 2023, per its Maven page) is a fallback to avoid in new code.

The API Interface and the Retrofit Instance

```
import retrofit2.http.GET

interface NotesApi {
    // suspend: Retrofit runs it off the main thread for you
    @GET("posts")
    suspend fun getNotes(): List<RemoteNoteDto>
}
```

```
import okhttp3.MediaType.Companion.toMediaType
import retrofit2.Retrofit
import retrofit2.converter.kotlinx.serialization.asConverterFactory

object NetworkModule {
    val notesApi: NotesApi by lazy {
        Retrofit.Builder()
            .baseUrl("https://jsonplaceholder.typicode.com/")       // https, so no clear
            .addConverterFactory(json.asConverterFactory("application/json".toMediaType(
            .build()
            .create(NotesApi::class.java)
    }
}
```

- `@GET("posts")` is appended to the base URL. The service used is a free test API that needs no key and returns pretend data; its address and fields were *not* checked against its documentation here (Chapter 6 flagged the same). Check it before relying on it.
- **HTTPS.** *(docs)* Cleartext HTTP is blocked by default from Android 9, so an `http://` address would fail. The docs say to allow cleartext only for specific domains through a network

security configuration, not app-wide, and not as a shortcut.

- **The converter** uses the shared `json` from Chapter 6, so `ignoreUnknownKeys` applies. *(blog sources)* If you mix converters, add this one last.
- **Main-safe.** *(docs, from a deprecated codelab)* Retrofit suspend functions are main-safe, so no `withContext` is needed here.

Changing the Entity: a Schema Change in Practice

A subtle problem: Room numbers local notes 1, 2, 3, and the server also has notes numbered 1, 2, 3. If you stored the server's ids as ours, the two sets would overwrite each other. So the entity gets a `remoteId` column, and a unique index on it so a second refresh replaces the same server note instead of duplicating it.

```
@Entity(
    tableName = "notes",
    indices = [Index(value = ["remoteId"], unique = true)]
)
data class NoteEntity(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    val text: String,
    val createdAt: Long = System.currentTimeMillis(),
    val remoteId: Long? = null          // the server's id, or null for a note made here
)

// In AppDatabase: raise the version (schema changed)
@Database(entities = [NoteEntity::class], version = 2, exportSchema = false)
```

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insertAll(notes: List<NoteEntity>)
```

```
// DTO to entity: the server's id goes in remoteId, our own id is generated
fun RemoteNoteDto.toEntity() = NoteEntity(text = title, remoteId = id)
```

(general knowledge) A unique index allows many rows whose `remoteId` is null, so notes made on the phone are not affected. Changing the entity means raising the database `version`, as

Chapter 4 said. While developing, the quickest fix is to **uninstall the app** so the old database is deleted; for a released app you would write a migration instead, so users' notes survive.

The Repository Tells the ViewModel What Happened

```
sealed interface RefreshResult {
    data object Success : RefreshResult
    data object NoConnection : RefreshResult
    data class ServerError(val code: Int) : RefreshResult
    data object BadReply : RefreshResult
}

// In NotesRepository
suspend fun refresh(): RefreshResult
```

```
import java.io.IOException
import kotlinx.serialization.SerializationException
import retrofit2.HttpException

class DefaultNotesRepository(
    private val noteDao: NoteDao,
    private val api: NotesApi
) : NotesRepository {

    // getNotesStream, addNote and deleteNote are unchanged: the UI still reads only fro

    override suspend fun refresh(): RefreshResult =
        try {
            val remote = api.getNotes()
            noteDao.insertAll(remote.map { it.toEntity() })     // write into the single
            RefreshResult.Success
        } catch (e: IOException) {
            RefreshResult.NoConnection
        } catch (e: HttpException) {
            RefreshResult.ServerError(e.code())
        } catch (e: SerializationException) {
            RefreshResult.BadReply
        }
}
```

Why a result type and not exceptions? Chapter 5's rule: the ViewModel must not know about the data layer's tools. If `refresh()` let `HttpException` (a Retrofit class) escape, the ViewModel would have to import Retrofit. Instead the repository turns each failure into a plain value. *(general knowledge)* A missing connection normally surfaces as an `IOException`, a non-success reply as an `HttpException` from Retrofit's suspend functions, and a reply that does not parse as a `SerializationException`; confirm what you actually get in Exercise 2. Only those three are caught, so cancellation (Chapter 3) still passes through.

```
val notesRepository: NotesRepository by lazy {
    DefaultNotesRepository(database.noteDao(), NetworkModule.notesApi)
}
```

```
// In FakeNotesRepository
override suspend fun refresh(): RefreshResult = RefreshResult.Success
```

Loading and Error States in the ViewModel and Screen

```
data class NotesUiState(
    val notes: List<Note> = emptyList(),
    val isLoading: Boolean = false,
    val isRefreshing: Boolean = false,       // new
    val userMessage: String? = null
)
```

```
private val isRefreshing = MutableStateFlow(false)

val uiState: StateFlow<NotesUiState> =
    combine(repository.getNotesStream(), message, isRefreshing) { notes, userMessage, re
        NotesUiState(
            notes = notes,
            isLoading = false,
            isRefreshing = refreshing,
            userMessage = userMessage
        )
    }.stateIn(
        scope = viewModelScope,
```

```
            started = SharingStarted.WhileSubscribed(5_000),
            initialValue = NotesUiState(isLoading = true)
        )

    fun refresh() {
        viewModelScope.launch {
            isRefreshing.value = true
            val result = repository.refresh()
            isRefreshing.value = false
            when (result) {
                RefreshResult.Success -> Unit
                RefreshResult.NoConnection -> message.value = "No connection"
                is RefreshResult.ServerError -> message.value = "Server error (${result.code
                RefreshResult.BadReply -> message.value = "Unexpected reply from the server"
            }
        }
    }
```

```
// In NotesScreen: a refresh button and a spinner
Row {
    Button(onClick = onRefresh, enabled = !state.isRefreshing) { Text("Refresh") }
    if (state.isRefreshing) CircularProgressIndicator()
}
```

The pattern is the same as Chapters 2 and 3: failures become a `userMessage` that the screen shows once; progress becomes a flag (`isRefreshing`). *(docs)* The recommendation to model UI state with a data class holding data, errors and loading signals is from Chapter 1's sources. The

WHAT HAPPENED	REPOSITORY RETURNS	USER SEES
Fetch worked	`Success`	The list updates by itself, because Room's Flow emits
No network	`NoConnection`	"No connection"; the saved notes still show
Server replied with an error	`ServerError(code)`	"Server error (404)"
Reply could not be read	`BadReply`	"Unexpected reply from the server"

Notice what is missing: after a successful refresh, the ViewModel does not copy the fetched notes into the state. It does not need to. The data went into Room, and the screen reads Room.

That is the single source of truth working, and it is why the app still shows its notes on an aeroplane.

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Architecture & Data 4 teaches REST calls, Gson or Moshi, error handling and loading states, with `LiveData` carrying the results. The calls are the same. What changes here: the JSON library is kotlinx.serialization (Chapter 6), results are plain values from the repository, and the network writes into Room rather than going straight to the screen.

Try It Yourself

EXERCISE 1

Add the permission, the dependencies, `NotesApi`, `NetworkModule` and the changes to the entity, DAO, repository, ViewModel and screen. Uninstall the app, run it, tap Refresh, then switch on aeroplane mode and restart. Report the converter coordinates you actually used and what you saw offline.

View solution

EXERCISE 2

Cause each failure on purpose: no network, a wrong path (for example `@GET("postz")`), and a URL that returns something that is not the expected JSON. Report the exception type you got each time and whether the repository mapped it as the chapter says.

View solution

EXERCISE 3

Make a refresh run automatically the first time the screen opens if the table is empty. Where should that logic live, and why not in the composable? Then explain, in your own words, why the screen reads Room and not the Retrofit reply.

View solution

NOT YET VERIFIED

Checked in the docs (summaries only): local database as the recommended source of truth; the repository fetching and persisting; Retrofit and Room suspend functions being main-safe (a deprecated codelab); cleartext off by default from Android 9; per-domain allowance through a network security configuration; `usesCleartextTraffic` being phased out.
Repositories and blogs only: `asConverterFactory` with a media type and a configured `Json`; the community converter's deprecation and its move into Retrofit.
General knowledge, not found in any search: the INTERNET permission; the first-party converter's Gradle coordinates, package and function name; the Retrofit annotations; which exceptions each failure produces; the unique-index design; the test service's address and fields.
Not verified: every listing and solution (unrun); the exercises (not run by me); the Room 3.0 form of the entity and DAO. Nothing was checked on your install.
Sources: Build an offline-first app (developer.android.com), Data layer (developer.android.com), Network security configuration (developer.android.com), Cleartext communications (developer.android.com), Get data from the internet codelab (developer.android.com), retrofit2-kotlinx-serialization-converter (GitHub, deprecated).

COMING UP

Chapter 8 adds DataStore: a small, safe place for settings such as a sort order or a dark-mode choice, replacing SharedPreferences.

Android Development with Compose: Architecture & Data

Course 5 · Chapter 8 · Remembering Settings

Room is for lists of records. For a handful of settings ("sort newest first", "dark mode", the last tab you used) there is a lighter tool: **DataStore**. It replaces the older `SharedPreferences`, and like everything else in this course it speaks coroutines and Flow.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked in the docs (one search, summaries only):** DataStore has two implementations, Preferences DataStore (key-value) and Proto DataStore (typed objects with protocol buffers); it is built on coroutines and Flow; the Preferences API exposes data as a Flow and has no `apply()` or `commit()`; Proto uses `updateData()` for an atomic read-modify-write; a `SharedPreferencesMigration` goes in the `produceMigrations` parameter of the `preferencesDataStore` delegate, with the old key names reused so values carry over; migrations run before any data is read or edited; reading DataStore inside a migration callback deadlocks; and the release page listed 1.3.0 as the latest stable version at the time of the search (the dependency below still uses a placeholder). **General knowledge, not checked:** the code shown (key functions, `edit` usage, the `catch` for `IOException`, the delegate defined at top level), "one instance per file", the comparison with `SharedPreferences`, and the DataStore-or-Room advice. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

DataStore or Room?

	DATASTORE	ROOM
Fits	A few small values: settings, flags, a last-used choice *(general knowledge)*	Lists of records, queries, relationships (Chapter 4)
Shape	Keys and values (Preferences) or one typed object (Proto) *(docs)*	Tables and rows
Reading	A `Flow` *(docs)*	A `Flow` from a DAO
Queries	None: you read the whole thing	SQL

And `rememberSaveable` (Android Development Fundamentals with Compose 4) is for state that only has to survive rotation on one screen, not forever.

Setup and Creating the DataStore

```
// Module build.gradle.kts (version is a placeholder)
dependencies {
    implementation("androidx.datastore:datastore-preferences:<datastore-version>")
}
```

```
import android.content.Context
import androidx.datastore.preferences.preferencesDataStore

// Top level, once per file name: one DataStore instance per file
val Context.settingsDataStore by preferencesDataStore(name = "settings")
```

(general knowledge) The delegate is declared **once, at the top level of a file**, as an extension on `Context`. Creating two DataStores for the same file name is an error in the library; this declaration is what guarantees one.

A Settings Repository

```
import androidx.datastore.core.DataStore
import androidx.datastore.preferences.core.Preferences
import androidx.datastore.preferences.core.booleanPreferencesKey
import androidx.datastore.preferences.core.edit
import androidx.datastore.preferences.core.emptyPreferences
import java.io.IOException
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.catch
import kotlinx.coroutines.flow.map

class SettingsRepository(
    private val dataStore: DataStore<Preferences>
) {
    private val newestFirstKey = booleanPreferencesKey("newest_first")

    // Reading: a Flow, so the UI follows changes automatically
    val newestFirst: Flow<Boolean> = dataStore.data
        .catch { e ->
            if (e is IOException) emit(emptyPreferences()) else throw e
```

```
        }
        .map { prefs -> prefs[newestFirstKey] ?: true }      // default when never set

    // Writing: a suspend function, no apply() or commit()
    suspend fun setNewestFirst(value: Boolean) {
        dataStore.edit { prefs -> prefs[newestFirstKey] = value }
    }
}
```

- **Reading is a `Flow`.** *(docs)* The Preferences API exposes data as a Flow, so a screen that collects it updates when the setting changes, from anywhere.
- **Writing is `edit`.** *(docs)* There is no `apply()` or `commit()`; `edit` is a suspend function, so call it from a coroutine.
- **A default for "never set".** *(general knowledge)* A missing key reads as `null`; `?: true` is the default.
- **The `catch`.** *(general knowledge)* Reading can fail with an `IOException` if the file cannot be read; this falls back to empty preferences, so defaults apply, rather than crashing.
- It is a **repository** in the sense of Chapter 5: the ViewModel never touches `DataStore` or key objects directly.

Wiring It Into the App

```
class NotesApp : Application() {
    // ...database and notesRepository as before...

    val settingsRepository: SettingsRepository by lazy {
        SettingsRepository(settingsDataStore)       // the Context extension from above
    }
}
```

```
data class NotesUiState(
    val notes: List<Note> = emptyList(),
    val newestFirst: Boolean = true,           // new
    val isLoading: Boolean = false,
    val isRefreshing: Boolean = false,
```

```
    val userMessage: String? = null
)
```

```
class NotesViewModel(
    private val repository: NotesRepository,
    private val settings: SettingsRepository
) : ViewModel() {

    private val message = MutableStateFlow<String?>(null)
    private val isRefreshing = MutableStateFlow(false)

    val uiState: StateFlow<NotesUiState> =
        combine(repository.getNotesStream(), settings.newestFirst, message, isRefreshing
            notes, newestFirst, userMessage, refreshing ->
            NotesUiState(
                notes = if (newestFirst) notes.sortedByDescending { it.createdAt }
                        else notes.sortedBy { it.createdAt },
                newestFirst = newestFirst,
                isRefreshing = refreshing,
                userMessage = userMessage
            )
        }.stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5_000),
            initialValue = NotesUiState(isLoading = true)
        )

    fun setNewestFirst(value: Boolean) {
        viewModelScope.launch { settings.setNewestFirst(value) }
    }

    // addNote, deleteNote, refresh and messageShown are as in the earlier chapters
}

// In the Factory: NotesViewModel(app.notesRepository, app.settingsRepository)
```

```
// In NotesScreen
Row(verticalAlignment = Alignment.CenterVertically) {
    Text("Newest first")
```

```
        Switch(checked = state.newestFirst, onCheckedChange = onNewestFirstChange)
    }
```

`combine` with a fourth flow is the same technique as Chapters 3 and 7: when the setting changes, the state rebuilds and the list re-sorts; no other code needs to know. The route passes `onNewestFirstChange = viewModel::setNewestFirst`. Toggle the switch, close the app completely and reopen it: the choice is still there.

Moving Over From SharedPreferences

```
import androidx.datastore.migrations.SharedPreferencesMigration

val Context.settingsDataStore by preferencesDataStore(
    name = "settings",
    produceMigrations = { context ->
        listOf(SharedPreferencesMigration(context, "old_prefs_file_name"))
    }
)
```

- *(docs)* A `SharedPreferencesMigration` goes in `produceMigrations`; reuse the old key names so values carry over.
- *(docs)* Migrations run before any data access, so the first read waits for them to finish.
- *(docs)* Do not read the DataStore inside a migration callback: it deadlocks, because DataStore returns nothing until all migrations complete.
- *(docs)* Migrations should be idempotent (safe to run more than once), and if the old file ends up empty the migration tries to delete it.

This app has no old `SharedPreferences` file, so the snippet is for reference. *(general knowledge)* Older code often used it to remember small values; `SharedPreferences` has a synchronous API, which is one reason it was replaced.

Proto DataStore, Described

(docs) The other flavour stores a **typed object** defined with protocol buffers instead of loose keys, and changes it with `updateData()`, an atomic read-modify-write. The Proto codelab

suggests a sentinel enum value such as `UNSPECIFIED` as the default, so a migration can tell whether a value still needs to be copied from `SharedPreferences`. *(general knowledge)* It needs a protobuf schema file and the protobuf Gradle plugin, so it is more setup; it is worth it when settings are a structured object and you want the compiler to check the types. This chapter does not build one.

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Architecture & Data 7 teaches DataStore as a replacement for `SharedPreferences`, covering Preferences and Proto DataStore. The ideas are identical. The Compose difference is how a setting reaches the screen: a `Flow` feeds the ViewModel's state, and the switch is just another part of `NotesUiState`.

Try It Yourself

EXERCISE 1

Add the dependency, `settingsDataStore`, `SettingsRepository` and the "Newest first" switch. Toggle it, kill the app, reopen it, and report whether the setting survived and whether the list re-sorted.

View solution

EXERCISE 2

Add a second setting, an integer limit on how many notes to show, using an `Int` key with a default. Show it with a simple control of your choice and apply it in the ViewModel. Report anything that surprised you.

View solution

EXERCISE 3

For each of these, say whether you would use `rememberSaveable`, a ViewModel, DataStore or Room, and why: the text typed into the new-note box; the current list of notes; the sort order; a dark-mode choice; a cache of 5,000 articles.

View solution

NOT YET VERIFIED

Checked in the docs (one search, summaries only): the two DataStore flavours; coroutines and Flow; no `apply()` or `commit()`; `updateData()` for Proto; `SharedPreferencesMigration` in `produceMigrations`; reusing key names; migrations run before data access; the deadlock warning; idempotent migrations and old-file deletion; the `UNSPECIFIED` sentinel suggestion; 1.3.0 listed as latest stable.

General knowledge, not checked: the code as shown (key functions, `edit`, the `IOException` catch, the top-level delegate); one instance per file; the `SharedPreferences` comparison; the DataStore, Room and `rememberSaveable` advice; Proto setup needs.

Not verified: every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.

Sources: DataStore (developer.android.com), Working with Preferences DataStore codelab (developer.android.com), Working with Proto DataStore codelab (developer.android.com), SharedPreferencesMigration API reference (developer.android.com), DataStore releases (developer.android.com).

COMING UP

The next chapter, Chapter 9 (Manual Dependency Injection), looks at what has been happening all along: the app container, the hand-written factory and the objects handed to the ViewModel, and how to organise that deliberately.

Android Development with Compose: Architecture & Data

Course 5 · Chapter 9 · Handing Objects Over

You have been doing dependency injection since Chapter 5 without the name. The ViewModel does not build its repository: someone hands it one. This chapter names the idea, gathers the scattered wiring into one deliberate place, and shows what it costs, which is the reason Chapter 10 exists.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked in the docs (one search, summaries only):** manual DI with a central `AppContainer` on the `Application` class that builds dependencies by hand, and the note that you manage it yourself; the framework does not let you pass values into a ViewModel constructor by default, so a `ViewModelProvider.Factory` is needed; the `viewModelFactory` and `initializer` builder reading from the container through `APPLICATION_KEY`; the factory passed to `viewModel(factory = ...)` in Compose; constructor injection recommended; testing by passing fakes or mocks; and Hilt or manual DI both recommended for simple apps, with Hilt generating the factory. **The docs flagged:** some pages are old or deprecated and the factory APIs have changed across ViewModel versions. **General knowledge, not checked:** the exact code and imports below, the object scoping remarks, and the `TimeProvider` example. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

The Problem

```
// A ViewModel that builds everything it needs itself
class NotesViewModel(app: Application) : ViewModel() {
    private val database = Room.databaseBuilder(app, AppDatabase::class.java, "notes.db"
    private val repository = DefaultNotesRepository(database.noteDao(), NetworkModule.no
    // ...
}
```

- **It cannot be tested** without a real Android `Application`, a real database and the network.
- **It cannot be changed** without editing the ViewModel: a different repository means a different ViewModel.
- **It hides what it needs.** You cannot see the dependencies by reading the constructor.

- **It may create two databases.** If two classes each build their own, they disagree (Chapter 4 said: one database for the whole app).

(general knowledge) These four points are the usual case for dependency injection; they are not quoted from the docs.

Constructor Injection

```
// The same ViewModel, asking for what it needs
class NotesViewModel(
    private val repository: NotesRepository,
    private val settings: SettingsRepository
) : ViewModel() {
    // ...
}
```

(docs) The architecture recommendations say to use dependency injection best practices, mainly **constructor injection** where possible: a class declares what it needs as constructor parameters and does not build them. **Injection** just means the objects are passed in from outside. Nothing magic, and no library, is required.

An Application Container

Somebody still has to build the objects. *(docs)* The manual DI guide suggests an `AppContainer`, created in the `Application` class, that builds the dependencies by hand. Chapters 5 to 8 had a growing set of `by lazy` properties on `NotesApp` and a `NetworkModule` object; this puts them in one tidy class:

```
import android.content.Context
import androidx.room.Room
import okhttp3.MediaType.Companion.toMediaType
import retrofit2.Retrofit
import retrofit2.converter.kotlinx.serialization.asConverterFactory

// What the app can supply
interface AppContainer {
    val notesRepository: NotesRepository
```

```
    val settingsRepository: SettingsRepository
}

// The one place that knows how everything is built
class DefaultAppContainer(private val context: Context) : AppContainer {

    private val database: AppDatabase by lazy {
        Room.databaseBuilder(context, AppDatabase::class.java, "notes.db").build()
    }

    private val notesApi: NotesApi by lazy {
        Retrofit.Builder()
            .baseUrl("https://jsonplaceholder.typicode.com/")
            .addConverterFactory(json.asConverterFactory("application/json".toMediaType(
            .build()
            .create(NotesApi::class.java)
    }

    override val notesRepository: NotesRepository by lazy {
        DefaultNotesRepository(database.noteDao(), notesApi)
    }

    override val settingsRepository: SettingsRepository by lazy {
        SettingsRepository(context.settingsDataStore)
    }
}
```

```
class NotesApp : Application() {
    lateinit var container: AppContainer

    override fun onCreate() {
        super.onCreate()
        container = DefaultAppContainer(this)
    }
}
```

- **An interface and a default implementation.** The interface is what the app *can supply*; the class is *how*. A test can supply a different implementation *(general knowledge)*.
- `by lazy` *(general knowledge)* builds each object the first time it is used, and exactly once, so there is one database and one Retrofit instance for the whole app.

- **The container is the only place that knows the full wiring.** The ViewModel, the repositories and the screens know only their own parameters.
- The manifest still names the class: `android:name=".NotesApp"` on the `application` element.

The ViewModel Factory

(docs) The framework does not let you pass your own values to a ViewModel's constructor by default; you supply a `ViewModelProvider.Factory` that knows how to build it. The `viewModelFactory` and `initializer` builder reads the container through `APPLICATION_KEY`. This replaces the factory that lived inside `NotesViewModel` in Chapter 5:

```
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.viewmodel.CreationExtras
import androidx.lifecycle.viewmodel.initializer
import androidx.lifecycle.viewmodel.viewModelFactory

object AppViewModelProvider {
    val Factory: ViewModelProvider.Factory = viewModelFactory {
        initializer {
            NotesViewModel(
                notesApplication().container.notesRepository,
                notesApplication().container.settingsRepository
            )
        }
    }
}

fun CreationExtras.notesApplication(): NotesApp =
    this[ViewModelProvider.AndroidViewModelFactory.APPLICATION_KEY] as NotesApp
```

```
@Composable
fun NotesRoute(viewModel: NotesViewModel = viewModel(factory = AppViewModelProvider.Facto
    // ...as before
}
```

(docs) The factory is passed to `viewModel(factory = ...)`. The docs also warn that these factory APIs have changed over the years, so check the current ViewModel documentation if an import

or call here does not compile.

Testing Gets Easy

```
// A plain JVM test: no container, no Android, no database
@Test
fun emptyNoteShowsMessage() = runTest {
    val viewModel = NotesViewModel(FakeNotesRepository(), FakeSettingsRepository())
    viewModel.addNote("   ")
    assertEquals("A note cannot be empty", viewModel.uiState.value.userMessage)
}
```

(docs) The guides say dependency injection lets you swap implementations, passing fakes or mocks to a ViewModel under test, and the Dagger guide shows exactly this: constructing the ViewModel directly with a fake repository and no DI framework. That is the payoff of everything since Chapter 5: the test above needs no container at all. `FakeSettingsRepository` only exists if `SettingsRepository` is an interface; Exercise 3 asks you to extract it.

What Manual DI Costs

COST	WHERE YOU FEEL IT
Boilerplate	Every new ViewModel needs a line in the factory; every new dependency a property in the container *(general knowledge)*
You manage the container yourself	*(docs)* The guide notes you create instances for all dependencies by hand and manage them yourself
Lifetimes are your problem	Which objects live for the whole app, which for one screen, is something you decide and enforce by hand *(general knowledge)*
Wiring mistakes show up late	A wrongly built object fails when it is first used, not when the app is built *(general knowledge)*

(docs) For a small app this is fine; the recommendations name manual DI and Hilt as the two options for simple apps. As the number of classes grows, the container and factory grow with it, and that is the problem Hilt solves. Seeing the hand-written version first means that when Chapter 10 generates it for you, you will know what it is generating.

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Architecture & Data 5 introduces dependency injection and goes straight to Hilt. This course puts manual injection first: the idea is the same, and Hilt in Chapter 10 then replaces the container and factory in this chapter with annotations.

Try It Yourself

EXERCISE 1

Create `AppContainer` and `DefaultAppContainer`, change `NotesApp`, add `AppViewModelProvider`, and remove the old factory and `NetworkModule`. Run the app and confirm nothing visible changed. Report any build errors, especially from the factory imports.

View solution

EXERCISE 2

Add a `TimeProvider` that `DefaultNotesRepository` uses for `createdAt`, supply it from the container, and write a test that proves a new note gets a fixed time. Say what you gained over calling `System.currentTimeMillis()` directly.

View solution

EXERCISE 3

Extract an interface from `SettingsRepository`, add a fake, and write the ViewModel test from the chapter. Then write a `FakeAppContainer` that supplies both fakes, and say where in the app it could be used.

View solution

NOT YET VERIFIED

Checked in the docs (one search, summaries only): the `AppContainer` on `Application`; managing it yourself; the factory requirement and `ViewModelProvider.Factory`; the `viewModelFactory` and `initializer` builder with `APPLICATION_KEY`; `viewModel(factory = ...)`; constructor injection;

testing with fakes or mocks; manual DI or Hilt for simple apps; Hilt generating the factory; the warning that some pages are old and the APIs have changed.

General knowledge, not checked: the code and imports as written; the four problems list; `by lazy` giving one instance; lifetimes and late wiring failures; the `TimeProvider` example.

Not verified: every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.

Sources: Manual dependency injection (developer.android.com), Create ViewModels with dependencies (developer.android.com), Add repository and Manual DI codelab (developer.android.com), Recommendations for Android architecture (developer.android.com), Using Dagger in Android apps (developer.android.com).

COMING UP

Chapter 10 introduces Hilt: what it generates, how to set it up with the current plugin and KSP, and how to turn this chapter's container and factory into a few annotations.

Android Development with Compose: Architecture & Data

Course 5 · Chapter 10 · Letting a Library Do the Wiring

Chapter 9 hand-wrote an `AppContainer` and a ViewModel factory. **Hilt** is Android's dependency injection library: you mark what exists and what is needed with annotations, and Hilt *generates* the container and the factories at build time. Everything from Chapter 9 still applies: constructor injection, fakes in tests. Hilt just removes the typing.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked in the docs (one search, summaries only): Hilt now supports KSP and the compiler dependency goes in the `ksp` configuration; the Hilt Gradle plugin is declared in the root build file with `apply false` and applied in each module, together with the KSP plugin; the Dagger docs listed Hilt 2.60.1 and the Android guide 2.57.1 (so versions here are placeholders); **the docs name the compiler artifact `hilt-android-compiler` in one snippet and `hilt-compiler` in another**, so check which your version uses; `hiltViewModel()` retrieves a ViewModel scoped to the current navigation destination, with `androidx.hilt:hilt-lifecycle-viewmodel-compose` (1.3.0 in the guide); testing needs `hilt-android-testing` and compilers on the test configurations; and the module that compiles your `Application` class needs all Hilt modules and injected classes in its dependencies. **The search did not cover the annotations in depth.** `@Provides`, `@Binds`, `@InstallIn`, `@Singleton`, `@HiltViewModel` and `@Inject` as written below are general knowledge, apart from one older deprecated codelab that used `SingletonComponent`. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

What Hilt Replaces

CHAPTER 9 (BY HAND)	HILT
`DefaultAppContainer` building each object	Modules with `@Provides` and `@Binds`, plus `@Inject` constructors
`NotesApp` creating the container	`@HiltAndroidApp` on `NotesApp`
`by lazy` for one instance	`@Singleton`
`AppViewModelProvider` factory	`@HiltViewModel` and `hiltViewModel()` *(docs: Hilt generates the factory)*

CHAPTER 9 (BY HAND)	HILT
Wiring mistakes found at first use	Missing bindings reported when the app is built *(general knowledge)*

Setup

```
// Root build.gradle.kts (versions are placeholders; take current ones from the Hilt doc
plugins {
    id("com.google.dagger.hilt.android") version "<hilt-version>" apply false
    id("com.google.devtools.ksp") version "<ksp-version>" apply false
}
```

```
// Module build.gradle.kts
plugins {
    id("com.google.devtools.ksp")
    id("com.google.dagger.hilt.android")
}

dependencies {
    implementation("com.google.dagger:hilt-android:<hilt-version>")
    ksp("com.google.dagger:hilt-compiler:<hilt-version>")    // the docs also show the n

    implementation("androidx.hilt:hilt-lifecycle-viewmodel-compose:<androidx-hilt-versio
}
```

- *(docs)* The plugin is declared at the root with `apply false` and applied in each module; the compiler goes through `ksp`. Room in Chapter 4 already needed KSP, so the plugin is probably already there.
- *(docs)* Older setups used `kapt`, which needs extra settings; avoid it in a new project.
- *(docs, unresolved)* Compiler artifact name: `hilt-compiler` or `hilt-android-compiler`. Take the one the current Hilt setup page for your version shows.
- Versions: take current ones from the Hilt release notes. The two docs pages above disagreed, which is a sign they are updated often.

The Entry Points

```
import android.app.Application
import dagger.hilt.android.HiltAndroidApp

@HiltAndroidApp
class NotesApp : Application()      // the container, onCreate and lateinit are gone
```

```
import dagger.hilt.android.AndroidEntryPoint

@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    // onCreate and setContent as before
}
```

(general knowledge) `@HiltAndroidApp` tells Hilt to build the app-wide set of objects (the "container" of Chapter 9, generated). `@AndroidEntryPoint` lets an Activity (or Fragment) receive injected objects; here it also makes `hiltViewModel()` work for composables inside it.

A ViewModel Hilt Can Build

```
import androidx.lifecycle.ViewModel
import dagger.hilt.android.lifecycle.HiltViewModel
import javax.inject.Inject

@HiltViewModel
class NotesViewModel @Inject constructor(
    private val repository: NotesRepository,
    private val settings: SettingsRepository
) : ViewModel() {
    // the body is unchanged; the companion object Factory is gone
}
```

```
import androidx.hilt.lifecycle.viewmodel.compose.hiltViewModel   // check the exact pack

@Composable
```

```
fun NotesRoute(viewModel: NotesViewModel = hiltViewModel()) {
    // ...as before
}
```

(docs) `hiltViewModel()` retrieves a ViewModel scoped to the current navigation destination, using the `androidx.hilt` Compose artifact. *(general knowledge)* The ViewModel's constructor the default-argument `viewModel(factory = ...)` are all gone. The package of `hiltViewModel` has moved over time, so use the IDE's import suggestion rather than trusting the line above.

Telling Hilt How to Build Things

Classes you own can say "Hilt, you may construct me" with an `@Inject` constructor:

```
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class DefaultNotesRepository @Inject constructor(
    private val noteDao: NoteDao,
    private val api: NotesApi
) : NotesRepository {
    // unchanged
}

@Singleton
class SettingsRepository @Inject constructor(
    private val dataStore: DataStore<Preferences>
) {
    // unchanged
}
```

(general knowledge) `@Singleton` means one instance for the whole app, the job `by lazy` did in Chapter 9. Hilt can build `DefaultNotesRepository` because it can build or provide each constructor parameter. For the rest, you write a **module**:

```
import android.content.Context
import dagger.Binds
```

```
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

// @Provides: for things you cannot (or do not want to) construct with @Inject
@Module
@InstallIn(SingletonComponent::class)
object DataModule {

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

    @Provides
    fun provideNoteDao(database: AppDatabase): NoteDao = database.noteDao()

    @Provides
    @Singleton
    fun provideDataStore(@ApplicationContext context: Context): DataStore<Preferences> =
        context.settingsDataStore

    @Provides
    @Singleton
    fun provideNotesApi(): NotesApi =
        Retrofit.Builder()
            .baseUrl("https://jsonplaceholder.typicode.com/")
            .addConverterFactory(json.asConverterFactory("application/json".toMediaType(
            .build()
            .create(NotesApi::class.java)
}

// @Binds: "when someone asks for the interface, give them this implementation"
@Module
@InstallIn(SingletonComponent::class)
abstract class RepositoryModule {

    @Binds
    abstract fun bindNotesRepository(impl: DefaultNotesRepository): NotesRepository
}
```

- `@Provides` *(general knowledge)* is for objects you cannot put `@Inject` on: a Room database, a Retrofit service, a DataStore. The function body is the recipe; its parameters are more dependencies Hilt supplies.
- `@Binds` *(general knowledge)* says which implementation to use when someone asks for an interface: `NotesRepository` gets `DefaultNotesRepository`. This is how Chapter 5's interface choice becomes configuration, and where a test could swap in a fake.
- `@InstallIn(SingletonComponent::class)` *(general knowledge, and in an older deprecated codelab)* says these bindings live for the whole app.
- `@ApplicationContext` *(general knowledge)* asks Hilt for the application `Context`, which a long-lived object should use rather than an Activity.

That is the whole of `DefaultAppContainer`, moved into annotations. Delete `AppContainer`, `DefaultAppContainer`, `AppViewModelProvider` and the old `NotesApp` body.

Testing

```
// The unit test is unchanged: constructor injection means no Hilt is needed to build th
val viewModel = NotesViewModel(FakeNotesRepository(), FakeSettingsRepository())
```

(docs) Tests that use Hilt itself need `hilt-android-testing` and a compiler on the test configurations (`kspTest`, `kspAndroidTest`), plus a test Activity annotated with `@AndroidEntryPoint`. *(general knowledge)* But the ViewModel unit test from Chapters 5 and 9 does not change: it builds the ViewModel directly with fakes. That is the benefit of constructor injection first.

Manual DI or Hilt?

	MANUAL (CHAPTER 9)	HILT
Code to write	A container and a factory that grow with the app	Annotations and a few modules
Magic	None: you can read every line	Generated code; errors can be harder to read *(general knowledge)*

	MANUAL (CHAPTER 9)	HILT
Setup	None	Plugin, KSP, versions to keep in step
Fits	Small apps *(docs: manual DI or Hilt for simple apps)*	Apps with many classes and screens; the Catalogue project course uses it

There is no right answer for a notes app. The honest summary of the docs is that both are acceptable for simple apps, and Hilt earns its setup as the dependency graph grows.

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Architecture & Data 5 teaches why dependency injection helps, Hilt setup, modules and injecting into a ViewModel. The same ground, in a different order: this course did the manual version first (Chapter 9) and then Hilt, and uses `hiltViewModel()` in Compose in place of `by viewModels()`, and KSP in place of `kapt`.

Try It Yourself

EXERCISE 1

Convert the app from Chapter 9's container to Hilt: add the plugins and dependencies, annotate `NotesApp`, `MainActivity` and the ViewModel, write the modules, and delete the container and factory. Report the Hilt, KSP and compiler artifact names and versions you used, and any build error.

View solution

EXERCISE 2

Provide the `Json` instance from a module and use it in `provideNotesApi`. Then bind a second interface (the `TimeProvider` from Chapter 9, Exercise 2) with `@Binds`. Explain when you need `@Provides` and when `@Binds` is enough.

View solution

EXERCISE 3

Break it on purpose: remove the `@Binds` function, then restore it and remove `@AndroidEntryPoint` from `MainActivity`. Write down what each error says and whether it appeared at build time or at run time. Then say which approach, Chapter 9's or Hilt, you would choose for this app, and why.

View solution

NOT YET VERIFIED

Checked in the docs (one search, summaries only): KSP support and the `ksp` configuration; the root `apply false` plugin declaration applied in each module; versions 2.60.1 (Dagger docs) and 2.57.1 (Android guide); the compiler artifact name discrepancy; `hiltViewModel()` scoped to a navigation destination and the `androidx.hilt` Compose artifact; Hilt generating the ViewModel factory; the testing artifacts and configurations; the module that compiles `Application` needing all Hilt modules in its dependencies; `kapt` needing extra settings.

General knowledge, not checked: `@HiltAndroidApp`, `@AndroidEntryPoint`, `@HiltViewModel`, `@Inject`, `@Singleton`, `@Provides`, `@Binds`, `@ApplicationContext` and `@InstallIn` as written (only `SingletonComponent` appeared in an old deprecated codelab); the `hiltViewModel` import package; build-time reporting of missing bindings; the comparison table.

Not verified: every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.

Sources: Dependency injection with Hilt (developer.android.com), Use Hilt with other Jetpack libraries (developer.android.com), Hilt Gradle build setup (dagger.dev), Dagger KSP (dagger.dev), Hilt testing guide (developer.android.com), Hilt releases (developer.android.com), Using Hilt in your Android app codelab (developer.android.com, deprecated).

COMING UP

Chapter 11 is the capstone: a small notes app that uses everything in this course, with a checklist to build it from.

Android Development with Compose: Architecture & Data

Course 5 · Chapter 11 · Putting It All Together

You have met every part separately. This chapter asks you to build one small app that uses all of them together: **a notes app** that stores notes in Room, shows them through a ViewModel, remembers a sort order in DataStore, can fetch sample notes from a server, can export its notes as JSON, and wires its objects together properly. There is very little new teaching here; the point is the joins.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

No new docs check for this chapter. Everything it relies on was checked in Chapters 1 to 10 (see each chapter's Not Yet Verified box), and the same labels apply here: the architecture layers and state flow in Chapter 1; ViewModel and `StateFlow` in Chapter 2; coroutines, `stateIn` and cancellation in Chapter 3; Room in Chapter 4 (which has the Room 2.x against 3.0 choice); the repository in Chapter 5; serialization in Chapter 6; Retrofit in Chapter 7 (**whose converter coordinates are unconfirmed**); DataStore in Chapter 8; manual injection in Chapter 9; Hilt in Chapter 10. **Added here, general knowledge:** the `@Update`-style targeted query, the five-flow limit of the typed `combine` and grouping flows into one, the `AlertDialog`, and the checklist and problems table, which are a summary of what went wrong in earlier chapters, not a docs-checked list. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

What You Are Building

FEATURE	USES
Add, edit, delete and list notes	Room, repository, ViewModel, stateless screen (Chapters 1 to 5)
Search as you type	A query held in the ViewModel's state
Newest-first or oldest-first	DataStore setting feeding the state (Chapter 8)
Refresh sample notes from a server	Retrofit, the converter, Room as the source of truth (Chapters 6 and 7)
Export the notes as JSON	kotlinx.serialization (Chapter 6)
Objects wired deliberately	Manual container (Chapter 9) or Hilt (Chapter 10): your choice

The Shape of the App

```
com.example.notes/
├── NotesApp.kt                 (Application; @HiltAndroidApp if you chose Hilt)
├── MainActivity.kt
├── di/                         (modules, or AppContainer and the factory)
├── data/
│   ├── local/                  NoteEntity, NoteDao, AppDatabase          (Chapter 4)
│   ├── remote/                 NotesApi, RemoteNoteDto                   (Chapters 6 a
│   ├── settings/               SettingsRepository + DataStore            (Chapter 8)
│   ├── Note.kt                 the model and the mappers                 (Chapter 5)
│   └── NotesRepository.kt      interface, default implementation, fake   (Chapter 5)
└── ui/
    └── notes/
        ├── NotesUiState.kt     (Chapter 1)
        ├── NotesViewModel.kt   (Chapters 2, 3 and 7 to 9)
        └── NotesScreen.kt      route and stateless screen                (Chapters 1 a
```

```
Room (single source of truth)  ──Flow──►  NotesRepository  ──Flow──►  NotesViewModel  -
      ▲                                                                  ▲   │
      │ insertAll (Chapter 7)                        DataStore Flow        │   │ eve
Retrofit + serialization                             (sort order) ─────────┘   │ sea
      ▲                                                                        ▼
 the server                                                            ViewModel fun
```

State flows down the top row; events and the network come in from below. Nothing in the UI knows Room, Retrofit or DataStore exist (Chapter 5's table, *who may know what*).

Build It in Stages

STAGE	DO THIS
1. Storage	Entity, DAO and database (Chapter 4); the `Note` model, mappers and repository interface with a fake (Chapter 5). **Checkpoint:** the DAO test passes.
2. State holder	`NotesUiState`, the ViewModel exposing a `StateFlow` built from the repository's Flow (Chapters 1 to 3). **Checkpoint:** the ViewModel test passes against the fake.

STAGE	DO THIS
3. Screen	Route and stateless screen with add, delete and a list, plus previews (Chapters 1 and 2). **Checkpoint:** add a note, rotate, close and reopen: it is still there.
4. Settings, search and edit	The DataStore sort switch (Chapter 8), a search box, and editing a note's text. **Checkpoint:** the sort order survives a restart; search narrows the list as you type.
5. Network and export	Refresh from the server into Room (Chapter 7), and export the notes as JSON (Chapter 6). **Checkpoint:** aeroplane mode still shows the list, and a failed refresh shows a message.
6. Wiring	Hand-written container and factory (Chapter 9) *or* Hilt (Chapter 10). **Checkpoint:** no class builds its own dependencies.

Finish each checkpoint before moving on. If a later stage breaks an earlier checkpoint, that is information about a leaky boundary.

The New Pieces

Three small additions that earlier chapters did not cover. Editing a note, with a targeted query so that other columns are left alone:

```
// NoteDao: one new query. A targeted update keeps the other columns (such as remoteId) a
@Query("UPDATE notes SET text = :text WHERE id = :id")
suspend fun updateText(id: Long, text: String)

// NotesRepository and DefaultNotesRepository
suspend fun updateNoteText(id: Long, text: String)

override suspend fun updateNoteText(id: Long, text: String) {
    noteDao.updateText(id, text)
}
```

(general knowledge) The tempting alternative is `@Update` with `note.toEntity()`, but `Note` has no `remoteId`, so the mapper would reset it and an imported note would turn into a local one.

The state and a JSON export of the notes:

```
data class NotesUiState(
    val notes: List<Note> = emptyList(),
    val query: String = "",
    val newestFirst: Boolean = true,
    val isLoading: Boolean = false,
    val isRefreshing: Boolean = false,
    val userMessage: String? = null,
    val exportText: String? = null
)
```

```
import kotlinx.serialization.Serializable

@Serializable
data class NoteExport(val text: String, val createdAt: Long)

fun exportNotesAsJson(notes: List<Note>): String =
    json.encodeToString(notes.map { NoteExport(it.text, it.createdAt) })
```

```
// In NotesScreen: the export result in a dialog
state.exportText?.let { text ->
    AlertDialog(
        onDismissRequest = onExportShown,
        confirmButton = { TextButton(onClick = onExportShown) { Text("Close") } },
        text = { Text(text) }
    )
}
```

Here is the whole ViewModel, so you can see the earlier chapters joined. Notice the `Transient` class: the typed `combine` takes five flows *(general knowledge)*, so the short-lived bits (a message and the export text) are grouped into one flow.

```
@HiltViewModel                      // or no annotation, if you chose manual DI
class NotesViewModel @Inject constructor(
    private val repository: NotesRepository,
    private val settings: SettingsRepository
) : ViewModel() {

    // Five flows is the limit of the typed combine: group the small, transient bits int
    private data class Transient(val message: String? = null, val exportText: String? =
```

```
    private val transient = MutableStateFlow(Transient())
    private val query = MutableStateFlow("")
    private val isRefreshing = MutableStateFlow(false)

    val uiState: StateFlow<NotesUiState> =
        combine(
            repository.getNotesStream(), settings.newestFirst, query, transient, isRefres
        ) { notes, newestFirst, q, t, refreshing ->
            val shown = notes.filter { it.text.contains(q, ignoreCase = true) }
            NotesUiState(
                notes = if (newestFirst) shown.sortedByDescending { it.createdAt }
                        else shown.sortedBy { it.createdAt },
                query = q,
                newestFirst = newestFirst,
                isRefreshing = refreshing,
                userMessage = t.message,
                exportText = t.exportText
            )
        }.stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5_000),
            initialValue = NotesUiState(isLoading = true)
        )

    fun onQueryChange(value: String) { query.value = value }

    fun setNewestFirst(value: Boolean) {
        viewModelScope.launch { settings.setNewestFirst(value) }
    }

    fun addNote(text: String) {
        if (text.isBlank()) { showMessage("A note cannot be empty"); return }
        viewModelScope.launch { repository.addNote(text) }
    }

    fun updateNote(id: Long, text: String) {
        if (text.isBlank()) { showMessage("A note cannot be empty"); return }
        viewModelScope.launch { repository.updateNoteText(id, text) }
    }

    fun deleteNote(note: Note) {
        viewModelScope.launch { repository.deleteNote(note) }
    }
```

```
    fun refresh() {
        viewModelScope.launch {
            isRefreshing.value = true
            val result = repository.refresh()
            isRefreshing.value = false
            when (result) {
                RefreshResult.Success -> Unit
                RefreshResult.NoConnection -> showMessage("No connection")
                is RefreshResult.ServerError -> showMessage("Server error (${result.code
                RefreshResult.BadReply -> showMessage("Unexpected reply from the server"
            }
        }
    }

    fun exportNotes() {
        viewModelScope.launch {
            val notes = repository.getNotesStream().first()
            transient.update { it.copy(exportText = exportNotesAsJson(notes)) }
        }
    }

    fun exportShown() = transient.update { it.copy(exportText = null) }
    fun messageShown() = transient.update { it.copy(message = null) }

    private fun showMessage(text: String) = transient.update { it.copy(message = text) }
}
```

Three Kinds of Test

```
// 1. DAO (androidTest, in-memory Room, Chapter 4)
@Test fun insertedNoteIsObserved() = runTest { /* as in Chapter 4 */ }

// 2. ViewModel (JVM, fakes, Chapters 5 and 9)
@Test fun emptyNoteShowsMessage() = runTest {
    val viewModel = NotesViewModel(FakeNotesRepository(), FakeSettingsRepository())
    viewModel.addNote("   ")
    assertEquals("A note cannot be empty", viewModel.uiState.value.userMessage)
}

// 3. Serialization (JVM, Chapter 6)
@Test fun exportContainsTheText() {
    val text = exportNotesAsJson(listOf(Note(1, "Hi", 5L)))
```

```
    assertTrue(text.contains("Hi"))
}
```

Acceptance Checklist

Your capstone is done when every line is true:

☐ The UI never imports a DAO, an entity, Retrofit or DataStore.

☐ Every screen state can be previewed without running the app.

☐ The list survives rotation, closing the app and aeroplane mode.

☐ A failed refresh shows a message and does not clear the list.

☐ The sort order survives a restart.

☐ One database, one Retrofit instance, one DataStore for the whole app.

☐ No loading starts in a ViewModel's `init`.

☐ Cancellation is never swallowed (`CancellationException` is rethrown if you catch broadly).

☐ The ViewModel can be built in a test with fakes and no Android.

☐ All versions are current and the build is clean.

When Something Goes Wrong

SYMPTOM	LIKELY CAUSE
Build fails in a generated Room or Hilt class	Wrong or mismatched plugin, KSP or library versions (Chapters 4 and 10); a typo in an annotation or query.
App crashes on first run after changing the entity	The database version was not raised or there is no migration; uninstall while developing (Chapter 7).
List is empty until the first refresh	Expected: the table starts empty; show a helpful empty state, or refresh when the table is empty (Chapter 7, Exercise 3).
Fetch works but notes overwrite each other	Server ids stored as local ids; use a separate `remoteId` with a unique index (Chapter 7).
State never updates after a change	The upstream flow is not being collected: with `WhileSubscribed` nothing runs without a collector (Chapter 3).

SYMPTOM	LIKELY CAUSE
Hilt reports a missing binding	An interface without a `@Binds`, or a class without an `@Inject` constructor (Chapter 10).

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Architecture & Data has no capstone chapter. This one is new, and it is the Compose-first course's answer to "how do these ten pieces fit in one app?". The larger worked project that applies all of this is Personal Catalogue: Android (Kotlin & Compose).

Try It Yourself

EXERCISE 1

Build stages 1 to 3: storage, the state holder and the screen. Show the checkpoint results: the DAO test and the ViewModel test passing, and a note that survives rotation and a restart.

View solution

EXERCISE 2

Build stages 4 and 5: the sort setting, search, editing, refresh from the server and the JSON export. Report which stage took longest, and which earlier chapter you had to reread.

View solution

EXERCISE 3

Build stage 6 your chosen way, then walk the acceptance checklist and write down any line that is not yet true. Finish with a short review: if the server went away for good, what would you have to change, and what could stay exactly as it is?

View solution

NOT YET VERIFIED

Checked in the docs: nothing new in this chapter. All docs checks are in Chapters 1 to 10, with the gaps and unresolved points noted there (the Room 2.x and 3.0 split, the unconfirmed Retrofit converter coordinates, the Hilt compiler artifact name, the `collectAsStateWithLifecycle` version note).

General knowledge, not checked: the targeted update query; the five-flow limit of the typed `combine`; the `AlertDialog` code; the file layout; the checklist and the problems table.

Not verified: the complete app has never been built or run, so the pieces from different chapters may need small fixes to compile together; every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.

Sources: the sources listed in the Not Yet Verified boxes of Android Development with Compose: Architecture & Data 1 to 10.

COURSE COMPLETE

You can now organise an app into layers, hold its state in a ViewModel, store and fetch its data, remember its settings and wire it together. Next in the Android track: Android Development with Compose: Production & Publishing (`android6`, not yet written), and Personal Catalogue: Android (Kotlin & Compose), a larger project that applies all of this.