ANDROID DEVELOPMENT

Architecture & Data

Jetpack Compose · ViewModel & State

Room Database · Retrofit Networking

Hilt Dependency Injection · Coroutines in Android

DataStore · MVVM Architecture

Philip Osztromok

Generated with Claude

Table of Contents

Android Development — Architecture & Data — 8 Chapters

🖌️ Jetpack Compose Intro

Course 1 built UI the traditional way: XML layouts plus imperative Kotlin code that finds Views and mutates them. Jetpack Compose replaces both halves of that with a single idea — UI described entirely as Kotlin functions, automatically redrawn when the data behind them changes. This chapter covers composables, state, recomposition, and how the mental model actually differs from everything in Course 1.

🧱 Composable Functions

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!")
}
```

A `@Composable` function *describes* a piece of UI — calling `Text(...)` doesn't create and configure a View object the way `binding.textView.text = "..."` did in Course 1; it declares "there should be text here, with this content." Composables are Kotlin functions in every technical sense — they take parameters, can call other composables, and can be as small or as large as any function — but by convention they're named with a capital letter, like a class, to visually mark them as UI-producing.

```
@Composable
fun ProfileCard(name: String, title: String) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(text = name, fontSize = 20.sp)
        Text(text = title, color = Color.Gray)
    }
}
```

`Column` and `Row` are Compose's layout composables — the direct replacements for Course 1's `LinearLayout` (vertical/horizontal orientation), just as Kotlin function calls instead of XML tags. `Modifier` is Compose's mechanism for spacing, sizing, and other per-element styling — the composable-world equivalent of an XML View's `layout_*` attributes, but chainable: `Modifier.padding(16.dp).fillMaxWidth()`.

🔄 State & Recomposition — The Core Idea

A composable function re-runs — **recomposes** — automatically whenever the state it reads changes, redrawing only what actually needs to change:

```
@Composable
fun Counter() {
  var count by remember { mutableStateOf(0) }

  Column {
    Text(text = "Count: $count")
    Button(onClick = { count++ }) {
      Text("Increment")
    }
  }
}
```

`remember { mutableStateOf(0) }` creates a piece of state that survives recomposition (though not, by default, an Activity being recreated — that's `rememberSaveable`, a small variant covered later). `by` is Kotlin Intermediate Chapter 4's property delegation again — `mutableStateOf` provides `getValue`/`setValue`, letting `count` be read and written like a plain `var`, while every read is secretly tracked by Compose.

⚠ This Is the Whole Trick: Compose Tracks Reads, Not Writes

When `count++` runs inside the button's `onClick`, Compose doesn't manually push a new value into the Text — it re-runs (*recomposes*) exactly the composable functions that **read** `count`, because it tracked that read the last time they ran. `Text(text = "Count: $count")` reads `count`, so it recomposes; if `Counter` had other composables that never touched `count`, those would be skipped entirely — an automatic, fine-grained optimization with no equivalent manual work required.

Compose vs XML/View System (Course 1)

	XML + Views (Course 1)	Jetpack Compose
UI defined in	Separate XML files	Kotlin functions
Updating the UI	binding.textView.text = "..." (imperative mutation)	Change state; recomposition updates the UI automatically
Finding a View	findViewById / view binding	N/A — no View objects to "find"
Reusability	A custom View class, or a <include> layout	Just a function — call it anywhere

Compose vs React — A Very Similar Shape

	React	Jetpack Compose
Component unit	A function component	A @Composable function
Local state	useState()	remember { mutableStateOf(...) }
Re-render trigger	State change → re-render	State change → recomposition
Optimization	Virtual DOM diffing	Skips composables whose inputs didn't change

If React (from the JavaScript courses) feels familiar, that's not a coincidence — both are declarative, state-driven UI systems built around the same core insight: describe what the UI should look like *given* the current state, and let the framework figure out the minimal work to get there, rather than hand-writing the update steps yourself.

👁️ @Preview — Seeing UI Without Running the App

A `@Preview`-annotated composable renders directly inside Android Studio's editor, without building, installing, or launching anything on an emulator:

```
@Preview(showBackground = true)
@Composable
fun ProfileCardPreview() {
    ProfileCard(name = "Philip", title = "Android Developer")
}
```

This is a genuinely fast feedback loop compared to Course 1's edit-build-run cycle for checking a layout change — a dedicated, no-argument preview function calling the real composable with sample data shows up right in the code editor's split view, updating live as the code changes.

 Wiring Compose Into an Activity

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ProfileCard(name = "Philip", title = "Android Developer")
        }
    }
}
```

Note two changes from Course 1: the base class is `ComponentActivity`, not `AppCompatActivity`, and `setContent { }` replaces `setContentView(R.layout.activity_main)` — there's no XML layout file at all for a Compose screen. Everything else from Course 1's Activity lifecycle (Chapter 2) still applies unchanged; Compose is a different way of building the *content* of a screen, not a different kind of Activity.

💻 Coding Challenges

Challenge 1: A Composable with Parameters

Write a composable `fun ProductCard(name: String, price: Double)` using a Column to display both values as separate Text elements, and a @Preview function calling it with sample data. Wire it into MainActivity via setContent { }.

Goal: Practice a basic parameterized composable and the Preview workflow.

→ Solution

Challenge 2: State and Recomposition

Write a composable `Counter()` with a remembered mutableStateOf Int starting at 0, a Text showing the current value, and two Buttons ("+" and "-") that increment/decrement it. Confirm the displayed count updates correctly when either button is tapped.

Goal: Practice remember/mutableStateOf and see recomposition happen for real.

→ Solution

Challenge 3: Composing Composables Together

Using Challenge 1's ProductCard, write a composable `ProductList()` that calls ProductCard three times (with different sample data) inside a Column, separated by some vertical spacing (Modifier or a Spacer). Add a @Preview for ProductList.

Goal: Practice composing smaller composables into a larger screen, the way small functions combine into larger ones.

→ Solution

💡 Course 1's XML Skills Aren't Wasted

Many real, existing Android codebases still use the View system from Course 1, and Compose screens frequently need to interoperate with legacy View-based code (via `AndroidView` or `ComposeView`) — understanding both isn't redundant, it's genuinely necessary for working on real production Android apps today. This course focuses on Compose going forward because it's what new code is built with, not because Course 1 was a detour.

🎯 What's Next

Next chapter: **ViewModel & State Management** — ViewModel lifecycle, LiveData, StateFlow, and UI state patterns.

🏛️ ViewModel & State Management

Last chapter's Counter kept its count inside remember { mutableStateOf(0) } — fine for a toy example, but that state resets on rotation (Course 1, Chapter 2), and it can't be shared between composables that aren't nested together. ViewModel solves both problems, and StateFlow (already covered fully in Kotlin Intermediate Chapter 2) is how it exposes state to the UI. This chapter is largely about combining tools already learned, not new syntax.

🤔 Why remember Isn't Always Enough

`remember` ties state to a composable's position in the UI tree — it survives recomposition, but not a configuration change like rotation, which (per Course 1, Chapter 2) can destroy and recreate the entire Activity. A `ViewModel` is deliberately scoped differently: it survives configuration changes automatically, and is destroyed only when its owning screen is genuinely, permanently gone (not just rotated).

Defining a ViewModel

```
class CounterViewModel : ViewModel() {
    private val _count = MutableStateFlow(0)
    val count: StateFlow<Int> = _count.asStateFlow()

    fun increment() {
        _count.value++
    }
}
```

The `private var _count` / `public val count` naming pattern is a deliberate encapsulation convention: the ViewModel exposes a **read-only** `StateFlow` externally, while keeping the **mutable** version private to itself — nothing outside the ViewModel can directly set `count`, only call `increment()`. This is the same underscore-prefix convention commonly used for backing properties, applied specifically to the StateFlow pattern from Kotlin Intermediate Chapter 2.

🔌 Using a ViewModel from Compose

```
@Composable
fun CounterScreen(viewModel: CounterViewModel = viewModel()) {
    val count by viewModel.count.collectAsStateWithLifecycle()

    Column(modifier = Modifier.padding(16.dp)) {
        Text(text = "Count: $count")
        Button(onClick = { viewModel.increment() }) {
            Text("Increment")
        }
    }
}
```

`viewModel = viewModel()` is a default parameter that fetches (or creates, on first call) the correctly-scoped `CounterViewModel` instance automatically — this is the mechanism that makes the ViewModel survive rotation, since the same instance is returned again after recreation rather than a fresh one being constructed. `collectAsStateWithLifecycle()` bridges a `StateFlow` into Compose's own state system, using `by` delegation exactly like `remember { mutableStateOf(...) }` did — the composable recomposes automatically whenever the ViewModel's `count` emits a new value.

> ⚠ Rotate the Emulator to See the Difference
>
> Swap Chapter 1's `Counter()` (using plain `remember`) for this chapter's `CounterScreen()` (using a ViewModel) and rotate the emulator after a few clicks — Chapter 1's count resets to 0; this chapter's survives. That side-by-side comparison, run for real, makes the entire reason ViewModel exists concrete rather than abstract.

📜 LiveData — Where You'll Still See It

Before `StateFlow` became idiomatic, `LiveData` was Android's standard observable state holder — it still appears constantly in existing codebases and some libraries, so it's worth recognizing even though new code generally prefers `StateFlow`:

```
// The LiveData equivalent of this chapter's StateFlow ViewModel
class CounterViewModel : ViewModel() {
    private val _count = MutableLiveData(0)
    val count: LiveData<Int> = _count

    fun increment() {
        _count.value = (_count.value ?: 0) + 1
    }
}
```

LiveData vs StateFlow

	LiveData	StateFlow
Lifecycle-aware?	Yes, built in	No — needs collectAsStateWithLifecycle() in Compose
Kotlin coroutines integration	Bolted on (asFlow())	Native — it IS a coroutines Flow type
Operators (map, combine, ...)	Limited	Full Flow operator set (Kotlin Intermediate Ch2)
Current status	Legacy, still widely used	Modern default for new code

📦 A UI State Pattern: One Data Class Per Screen

Rather than several separate loose `StateFlow`s (one for loading, one for data, one for an error message — easy to get into an inconsistent combination), a common pattern bundles a screen's entire state into **one** data class:

```
data class ProfileUiState(
    val isLoading: Boolean = false,
    val userName: String? = null,
    val errorMessage: String? = null
)

class ProfileViewModel : ViewModel() {
    private val _uiState = MutableStateFlow(ProfileUiState(isLoading = true))
    val uiState: StateFlow<ProfileUiState> = _uiState.asStateFlow()

    fun onLoadSuccess(name: String) {
        // copy() from Kotlin Fundamentals Chapter 4 — update just what changed
        _uiState.value = _uiState.value.copy(isLoading = false, userName = name)
    }
}
```

`_uiState.value.copy(...)` is Kotlin Fundamentals Chapter 4's data class `copy()`, doing real work here — every state update produces a new, complete, internally-consistent `ProfileUiState` instance rather than mutating individual fields piecemeal, which is exactly what makes it impossible to accidentally end up with `isLoading = true` and a populated `userName` at the same time.

ViewModel + StateFlow vs Frontend State Management

	React (Redux/Zustand-style)	Android ViewModel
Holds state outside the UI tree	A store	A ViewModel
Single state object per screen	A reducer's state shape	A UiState data class
Survives re-render / rotation	Store persists across re-renders inherently	ViewModel persists across configuration changes
Subscribing from UI	useSelector() / a store hook	collectAsStateWithLifecycle()

💻 Coding Challenges

Challenge 1: A ViewModel-Backed Counter

Rewrite Chapter 1's Counter composable to use a CounterViewModel (as shown in this chapter) instead of remember. Run it, increment several times, rotate the emulator, and confirm the count survives — unlike Chapter 1's version.

Goal: Directly experience the rotation-survival difference ViewModel provides.

→ Solution

Challenge 2: A UiState Data Class

Write a data class TodoUiState(val items: List<String> = emptyList(), val isLoading: Boolean = true) and a TodoViewModel exposing it as a StateFlow. Add a function addItem(text: String) that updates the state via copy(), appending the new item and setting isLoading to false.

Goal: Practice the single-data-class UI state pattern with copy()-based updates.

→ Solution

Challenge 3: Collecting UiState in a Composable

Write a TodoScreen composable that collects TodoUiState from Challenge 2's TodoViewModel using collectAsStateWithLifecycle(), shows a loading Text while isLoading is true, and otherwise shows each item using a Text per entry inside a Column, plus a Button that calls addItem() with sample text.

Goal: Practice the full ViewModel-to-Compose data flow, conditionally rendering based on state.

→ Solution

💡 Nothing Here Was Genuinely New Syntax

StateFlow, MutableStateFlow, and data class copy() were all fully covered in the Kotlin courses — this chapter's real content is the architectural pattern (ViewModel as the state owner, a single UiState per screen, Compose as a pure function of that state), not new language features. That's a deliberate sign the earlier courses did their job: Android-specific chapters increasingly become "apply what you already know" rather than "learn something brand new."

🎯 What's Next

Next chapter: **Room Database** — entities, DAOs, the database class, and coroutines with Room.

🗄️ Room Database

Room is Android's standard local database library — a layer over SQLite that turns raw SQL into typed Kotlin classes and functions, checked at compile time. This chapter covers the three pieces every Room setup needs (Entity, DAO, Database), and how they connect to the ViewModel/StateFlow pattern from Chapter 2 to keep the UI in sync with what's actually stored on disk.

📋 @Entity — A Table, as a Data Class

```
@Entity(tableName = "tasks")
data class Task(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val title: String,
    val isDone: Boolean = false
)
```

An `@Entity` is an ordinary data class (Kotlin Fundamentals Chapter 4) with annotations describing how it maps to a database table — each constructor property becomes a column, and `@PrimaryKey(autoGenerate = true)` means Room assigns each new row's `id` automatically, so `id = 0` on a new `Task` is just a placeholder before insertion.

🔌 @Dao — The Query Interface

A DAO (Data Access Object) is an `interface` — Room generates the actual implementation at compile time, from annotated method signatures and SQL:

```
@Dao
interface TaskDao {

    @Query("SELECT * FROM tasks ORDER BY id DESC")
    fun getAllTasks(): Flow<List<Task>>

    @Insert
suspend fun insert(task: Task)

    @Update
suspend fun update(task: Task)

    @Delete
suspend fun delete(task: Task)
}
```

Flow-Returning Queries — Live Data

`getAllTasks(): Flow<List<Task>>` doesn't run once — Room re-emits automatically whenever the underlying `tasks` table changes, so a collector always sees the current data without ever manually re-querying. This is Kotlin Intermediate Chapter 2's `Flow`, produced by Room itself.

suspend Functions — Coroutine-Native Writes

`insert`/`update`/`delete` are `suspend` functions (Kotlin Intermediate Chapter 1) — calling them from a coroutine automatically runs the actual disk I/O off the main thread, with none of the manual thread-management older Android database code needed.

⚠ Room Checks Your SQL at Compile Time

The string inside `@Query(...)` is genuinely validated against the `@Entity` classes it references while compiling — a typo'd column name or a query that doesn't match the DAO method's return type is a compile error, not something that only surfaces as a runtime crash. This is one of Room's biggest practical advantages over writing raw SQLite calls by hand.

🏛️ @Database — Tying It Together

```
@Database(entities = [Task::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
  abstract fun taskDao(): TaskDao

  companion object {
    @Volatile
private var INSTANCE: AppDatabase? = null
fun getDatabase(context: Context): AppDatabase {
      return INSTANCE ?: synchronized(this) {
        Room.databaseBuilder(context, AppDatabase::class.java, "app_database")
          .build()
          .also { INSTANCE = it }
      }
    }
  }
}
```

`@Database(entities = [Task::class], version = 1)` lists every entity the database contains, and a version number Room uses to detect when a schema migration is needed (covered more fully in a later production-focused chapter). The `companion object` singleton pattern (Kotlin Fundamentals Chapter 4) ensures the whole app shares one `AppDatabase` instance — creating a fresh one per screen would be wasteful and could cause data inconsistency between them.

🔄 Wiring Room Into a ViewModel

This is where Chapter 2's StateFlow pattern and this chapter's DAO Flow meet — a DAO's `Flow` converts directly into a ViewModel-exposed `StateFlow`:

```
class TaskViewModel(private val dao: TaskDao) : ViewModel() {

  val tasks: StateFlow<List<Task>> = dao.getAllTasks()
    .stateIn(
      scope = viewModelScope,
      started = SharingStarted.WhileSubscribed(5000),
      initialValue = emptyList()
    )

  fun addTask(title: String) {
    viewModelScope.launch {
      dao.insert(Task(title = title))
    }
  }
}
```

`viewModelScope` is a `CoroutineScope` (Kotlin Intermediate Chapter 1) tied to the ViewModel's own lifetime — any coroutine launched in it is automatically cancelled when the ViewModel is cleared, so `addTask`'s `launch` never needs manual cleanup. `.stateIn(...)` converts the DAO's cold `Flow` into a hot, shareable `StateFlow` — `WhileSubscribed(5000)` keeps it active for 5 seconds after the last collector disappears (surviving a brief rotation gap) before actually stopping the underlying query.

Room vs a Backend ORM

	Node.js + mysql2 (Node Course 2)	Room
Where the database lives	A remote server	On-device, embedded (SQLite)
Query validation	Runtime (bad SQL fails when it runs)	Compile time
Live-updating results	Not automatic — re-query manually	Flow-returning queries update automatically
Async model	Promises / async-await	Kotlin suspend functions / Flow

💻 Coding Challenges

Challenge 1: Entity, DAO, and Database

Define a Note entity (id, title, content), a NoteDao interface with a Flow-returning getAllNotes() query and a suspend insert() function, and an AppDatabase abstract class wiring them together with the companion-object singleton pattern.

Goal: Practice the three-piece Room setup from scratch.

→ Solution

Challenge 2: A ViewModel Backed by Room

Write a NoteViewModel that exposes Challenge 1's getAllNotes() as a StateFlow via stateIn (with viewModelScope and WhileSubscribed(5000)), and an addNote(title: String, content: String) function that inserts via viewModelScope.launch.

Goal: Practice connecting a DAO's Flow to a ViewModel-exposed StateFlow, and a suspend write via viewModelScope.

→ Solution

Challenge 3: A Query with a WHERE Clause

Add a DAO function fun searchNotesByTitle(query: String): Flow<List<Note>> using @Query with a SQL WHERE title LIKE :query clause (using SQLite's % wildcard). Add a comment explaining what happens if the parameter name in the method signature doesn't match the :placeholder name in the query string.

Goal: Practice a parameterized @Query beyond the simple SELECT * examples in the chapter.

→ Solution

💡 The Flow → StateFlow → Compose Chain Is the Whole Architecture

Room's Flow, the ViewModel's StateFlow, and Compose's collectAsStateWithLifecycle() form one continuous pipe: change the database, and the UI updates itself automatically, with no manual "refresh" step anywhere in that chain. This three-layer pattern — data layer emits Flow, ViewModel exposes StateFlow, UI collects it — is close to universal across real, modern Android apps, not specific to this course's example.

🎯 What's Next

Next chapter: **Networking with Retrofit** — REST calls, Gson/Moshi, error handling, and loading states.

🌐 Networking with Retrofit

Retrofit is the standard Android HTTP client — it turns a REST API into a Kotlin interface, converting JSON into typed data classes automatically. This chapter covers defining an API interface, JSON conversion, and wrapping the result in the same sealed-class pattern already seen in Kotlin Intermediate Chapter 2's Flow example, now used for real network error handling.

📡 Defining an API Interface

```
data class User(
    val id: Int,
    val name: String,
    val email: String
)

interface ApiService {
    @GET("users")
    suspend fun getUsers(): List<User>

    @GET("users/{id}")
    suspend fun getUser(@Path("id") id: Int): User
@POST("users")
    suspend fun createUser(@Body user: User): User
}
```

There's no implementation here at all — `ApiService` is just an interface with annotated method signatures; Retrofit generates the actual networking code behind it. `suspend fun` means each call is a genuine Kotlin Intermediate Chapter 1 suspend function — no callbacks, no manual thread-switching, and the JSON response comes back already converted into the declared return type.

🔄 JSON Conversion — Gson or Moshi

Retrofit itself doesn't parse JSON — a **converter** library does, plugged in as a converter factory:

```
// Moshi (the more modern, Kotlin-friendly choice)
data class User(
    val id: Int,
    val name: String,
    @Json(name = "email_address") val email: String // maps a different JSON key name to this property
)
```

Gson vs Moshi

	Gson	Moshi
Reflection-based	Yes	Optional — can use compile-time code generation instead
Kotlin null-safety awareness	Limited — can silently produce nulls for non-null types	Better — respects Kotlin nullability more strictly
Renaming a JSON field	@SerializedName("email_address")	@Json(name = "email_address")

Either works and both are widely used in real projects; Moshi is generally the more actively recommended choice for new Kotlin code specifically because of its stricter null-safety behavior, which matters given how central null safety is to the rest of this curriculum (Kotlin Fundamentals Chapter 3).

Building the Retrofit Instance

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://api.example.com/")
    .addConverterFactory(MoshiConverterFactory.create())
    .build()

val apiService: ApiService = retrofit.create(ApiService::class.java)
```

`retrofit.create(ApiService::class.java)` is Kotlin Intermediate Chapter 6's reflection in action, though wrapped away inside Retrofit's own library code — it dynamically generates a real implementation of the `ApiService` interface at runtime, wiring each annotated method to an actual HTTP call. Like the database singleton pattern from last chapter, one shared Retrofit instance (and one shared `ApiService`) is normal for an entire app, not recreated per screen.

⚠️ Error Handling — Reusing a Familiar Sealed Class

Kotlin Intermediate Chapter 2 introduced exactly this shape for a different reason (handling a Flow's possible states) — the same `sealed class` pattern is the standard way to represent "a network call that might succeed, fail, or still be in progress":

```
sealed class NetworkResult<out T>
data class Success<T>(val data: T) : NetworkResult<T>()
data class Error(val message: String) : NetworkResult<Nothing>()
object Loading : NetworkResult<Nothing>()

suspend fun fetchUsers(api: ApiService): NetworkResult<List<User>> {
    return try {
        NetworkResult.Success(api.getUsers())
    } catch (e: IOException) {
        NetworkResult.Error("No internet connection")
    } catch (e: HttpException) {
        NetworkResult.Error("Server error: ${e.code()}")
    }
}
```

`NetworkResult<out T>` uses Kotlin Intermediate Chapter 3's covariance — `out T` is what allows `Error` and `Loading` to both implement `NetworkResult<Nothing>` while still being treated as a `NetworkResult<List<User>>` (or any other `T`) where needed, since they never actually need to hold a real `T` value. `IOException` generally means "never reached the server" (no connection, DNS failure); `HttpException` means "reached the server, got back an error status code" — distinguishing them gives a more useful error message than one generic catch-all.

📲 Loading States in the ViewModel

This connects directly to Chapter 2's single-`UiState`-per-screen pattern:

```
data class UsersUiState(
    val isLoading: Boolean = false,
    val users: List<User> = emptyList(),
    val errorMessage: String? = null
)

class UsersViewModel(private val api: ApiService) : ViewModel() {
    private val _uiState = MutableStateFlow(UsersUiState())
    val uiState: StateFlow<UsersUiState> = _uiState.asStateFlow()

    init {
        loadUsers()
    }

    fun loadUsers() {
        viewModelScope.launch {
            _uiState.value = _uiState.value.copy(isLoading = true)
            when (val result = fetchUsers(api)) {
                is NetworkResult.Success ->
                    _uiState.value = _uiState.value.copy(isLoading = false, users = result.data)
                is NetworkResult.Error ->
                    _uiState.value = _uiState.value.copy(isLoading = false, errorMessage = result.message)
                NetworkResult.Loading -> Unit // not produced by fetchUsers here, but the when must stay exhaustive
            }
        }
    }
}
```

Every earlier architectural piece shows up together here: `init { }` (Kotlin Fundamentals Chapter 4) triggers the initial load, `viewModelScope.launch` (this chapter's Retrofit calls are suspend functions, needing a coroutine), and the exhaustive `when` (Kotlin Fundamentals Chapter 5's sealed class matching) ensures every possible `NetworkResult` case updates the UI state correctly.

Retrofit vs JavaScript fetch/axios

	JavaScript (fetch/axios)	Retrofit
Defining an endpoint	A URL string passed to fetch()/axios.get()	An annotated interface method
JSON parsing	response.json() (manual, untyped by default)	Automatic, into a typed data class (Gson/Moshi)
Async model	Promises / async-await	Kotlin suspend functions
Error handling	try/catch around fetch, checking response.ok	try/catch around the suspend call, per-exception-type

💻 Coding Challenges

Challenge 1: An API Interface

Define a data class Post(val id: Int, val title: String, val body: String) and an interface PostApiService with a suspend getPosts(): List<Post> GET endpoint and a suspend getPost(id: Int): Post GET endpoint using @Path. Build a Retrofit instance and PostApiService for https://jsonplaceholder.typicode.com/ using a converter factory of your choice.

Goal: Practice defining a Retrofit API interface and building the client instance.

→ Solution

Challenge 2: NetworkResult Error Handling

Write a suspend fun fetchPosts(api: PostApiService): NetworkResult<List<Post>> using the chapter's NetworkResult sealed class, catching both IOException and HttpException with distinct error messages.

Goal: Practice the try/catch-into-sealed-class error wrapping pattern.

→ Solution

Challenge 3: A Full ViewModel with Loading State

Write a PostsUiState data class (isLoading, posts, errorMessage) and a PostsViewModel that loads posts on init using Challenge 2's fetchPosts(), updating uiState correctly through loading, success, and error paths using an exhaustive when.

Goal: Practice the complete Retrofit-to-UiState pipeline end to end.

→ Solution

💡 Notice How Little Was Genuinely New

Suspend functions, sealed classes, covariance, data class copy(), reflection, property delegation — every one of them from earlier chapters, recombined for a new purpose. Networking in a real Android app isn't a separate skill so much as it's these same tools applied to yet another data source, joining local storage (Room, last chapter) as another Flow/StateFlow feeding the same UI pattern.

🎯 What's Next

Next chapter: **Dependency Injection with Hilt** — why DI matters, Hilt setup, modules, and injecting into a ViewModel.

💉 Dependency Injection with Hilt

Chapters 3 and 4 built a database singleton and a Retrofit singleton by hand, each with its own manual companion-object wiring. Hilt automates exactly that kind of setup — classes declare what they need, and Hilt constructs and hands it to them, rather than each class constructing its own dependencies. This chapter covers why that matters, and Hilt's four core building blocks.

🤔 The Problem, Concretely

Chapter 4's `UsersViewModel` needed an `ApiService` — but where does that instance actually come from? Without DI, every class needing it either constructs its own (duplicating the Retrofit setup everywhere) or reaches into some global singleton object directly:

```
// Without DI — the ViewModel reaches out and grabs its own dependency
class UsersViewModel : ViewModel() {
    private val api = NetworkModule.retrofit.create(ApiService::class.java)
    // ...
}
```

This works, but it hardcodes exactly which `ApiService` implementation `UsersViewModel` uses — there's no way to substitute a fake one for testing, and every class that needs an `ApiService` repeats this same construction line. **Dependency injection** flips the direction: a class declares what it needs as a constructor parameter, and something else — here, Hilt — is responsible for actually providing a real instance.

```
// With DI — the dependency is handed in, not fetched
@HiltViewModel
class UsersViewModel @Inject constructor(
    private val api: ApiService
) : ViewModel() { /* ... */ }
```

`UsersViewModel` no longer knows or cares how `ApiService` is built — a real one in production, a fake one in tests, both work identically as far as this class is concerned.

🏁 Hilt Setup — Two Annotations to Start

```
// A custom Application class — required for Hilt
@HiltAndroidApp
class MyApplication : Application()


// Every Activity that uses Hilt-injected things needs this
@AndroidEntryPoint
class MainActivity : ComponentActivity() { /* ... */ }
```

`@HiltAndroidApp` generates Hilt's base dependency container for the whole app — it must go on a custom `Application` subclass, registered in `AndroidManifest.xml`'s `<application android:name=".MyApplication">`. `@AndroidEntryPoint` marks an Activity (or Fragment) as able to receive Hilt-provided dependencies — without it, Hilt won't wire anything into that class at all.

📦 @Module + @Provides — For Things You Don't Own

Hilt can't automatically construct classes it doesn't control the source of — `Retrofit` and `RoomDatabase` come from external libraries, so a **module** tells Hilt explicitly how to build them:

```
@Module
@InstallIn(SingletonComponent::class)
object NetworkModule {

    @Provides
@Singleton
fun provideRetrofit(): Retrofit {
        return Retrofit.Builder()
            .baseUrl("https://jsonplaceholder.typicode.com/")
            .addConverterFactory(MoshiConverterFactory.create())
            .build()
    }

    @Provides
@Singleton
fun provideApiService(retrofit: Retrofit): ApiService {
        return retrofit.create(ApiService::class.java)
    }
}
```

`@InstallIn(SingletonComponent::class)` scopes these provisions to the whole app's lifetime; `@Singleton` on each function ensures only one instance is ever created and reused — this is precisely the manual `companion object` singleton pattern from Chapter 3, now handled declaratively. Notice `provideApiService` takes a `retrofit: Retrofit` parameter — Hilt automatically supplies it from `provideRetrofit` above, chaining dependencies together without any manual ordering.

🏷️ @Inject Constructor — For Things You Do Own

For a class you wrote yourself (like a repository wrapping the DAO and API calls together), no module is needed at all — just annotate the constructor:

```
class UserRepository @Inject constructor(
    private val api: ApiService,
    private val dao: TaskDao
) {
    suspend fun refreshUsers() { /* ... */ }
}
```

Hilt sees `@Inject constructor`, recognizes it can build a `UserRepository` by supplying an `ApiService` (from `NetworkModule`) and a `TaskDao` (assuming a similar `DatabaseModule` providing it), and does so automatically wherever a `UserRepository` is itself requested — no explicit "here's how to build a UserRepository" module is required, since Hilt can already see the whole recipe directly in the constructor.

🔌 Injecting Into a ViewModel

```
@HiltViewModel
class UsersViewModel @Inject constructor(
    private val repository: UserRepository
) : ViewModel() { /* ... */ }


// In a composable — replaces Chapter 1's viewModel() with hiltViewModel()
@Composable
fun UsersScreen(viewModel: UsersViewModel = hiltViewModel()) {
    // ...
}
```

`@HiltViewModel` plus `hiltViewModel()` (instead of Chapter 1's plain `viewModel()`) is the only change needed at the call site — everything about how the ViewModel survives rotation, scopes to the screen, and exposes StateFlow works identically to Chapter 2, just with its dependencies supplied automatically instead of hardcoded inside the class.

Manual Wiring vs Hilt — What Actually Changed

	Manual (Chapters 3-4)	Hilt
Database singleton	Hand-written companion object + synchronized block	@Provides @Singleton function
ViewModel construction	Passed a dao/api manually via a ViewModelFactory	@Inject constructor — Hilt supplies it
Swapping a real dependency for a test fake	Requires editing the class itself	Swap the @Module's binding — the class is unchanged

💻 Coding Challenges

Challenge 1: Application and Activity Setup

Add a custom Application class annotated @HiltAndroidApp, register it in AndroidManifest.xml, and annotate MainActivity with @AndroidEntryPoint. Add a comment explaining what would go wrong if @AndroidEntryPoint were forgotten on an Activity that needs a Hilt-injected ViewModel.

Goal: Practice the minimum Hilt setup every project needs.

→ Solution

Challenge 2: A Module for Room

Write a DatabaseModule (@Module, @InstallIn(SingletonComponent::class)) with @Provides @Singleton functions providing an AppDatabase (using Room.databaseBuilder with an applicationContext parameter) and a TaskDao (from that database's taskDao() function).

Goal: Practice writing a @Module for a dependency Hilt can't construct automatically, following the chapter's NetworkModule pattern for a different kind of dependency.

→ Solution

Challenge 3: A Repository and Injected ViewModel

Write a TaskRepository with an @Inject constructor taking a TaskDao, exposing a getAllTasks(): Flow<List<Task>> function that delegates to the DAO. Write a @HiltViewModel TaskViewModel with an @Inject constructor taking the repository, exposing its tasks as a StateFlow via stateIn (Chapter 3's pattern).

Goal: Practice the full chain: DAO → Repository → ViewModel, all wired by Hilt with no manual construction anywhere.

→ Solution

💡 The Payoff Is Mostly Invisible Until Testing

DI's benefit doesn't show up much in a single-screen toy app — the manual approach from Chapters 3-4 works fine at that scale. It pays off as an app grows (many classes sharing the same dependencies, without each repeating construction logic) and especially once automated testing enters the picture, where swapping a real network/database dependency for a fake one is exactly what DI is designed to make painless.

🎯 What's Next

Next chapter: **Coroutines in Android** — `viewModelScope`, `lifecycleScope`, and choosing between the IO and Main dispatchers in an Android-specific context.

⚙️ Coroutines in Android

viewModelScope has appeared in every chapter since Chapter 2, always used without much explanation of what it actually is or why it's safe to launch into. This chapter fills that gap — the Android-specific coroutine scopes, the repeatOnLifecycle pattern for safely collecting a Flow outside Compose, and where dispatchers actually matter in practice.

🔬 viewModelScope — What It Actually Is

`viewModelScope` is a `CoroutineScope` (Kotlin Intermediate Chapter 1) automatically provided to every `ViewModel`, tied precisely to that ViewModel's own lifetime:

```
class TaskViewModel(private val repository: TaskRepository) : ViewModel() {
    fun addTask(title: String) {
        viewModelScope.launch {
            repository.insert(Task(title = title))
        }
    }
    // No manual cleanup needed — when this ViewModel is cleared,
    // every coroutine launched via viewModelScope is cancelled automatically.
}
```

The ViewModel base class calls `viewModelScope.cancel()` internally inside its own `onCleared()` — a coroutine started with `viewModelScope.launch { }` gets structured-concurrency cancellation (Kotlin Intermediate Chapter 2) for free, tied to exactly the right lifetime: the ViewModel's, which (per Chapter 2 of this course) already survives rotation correctly.

🖼️ lifecycleScope — Tied to the Activity/Fragment Instead

Outside a ViewModel — directly in an Activity or Fragment, most often in code interoperating with the older View system from Course 1 — `lifecycleScope` plays the equivalent role, tied to that specific Activity/Fragment instance's lifecycle (Course 1, Chapters 2 and 6):

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        lifecycleScope.launch {
            // Cancelled automatically when this Activity is destroyed
val data = repository.fetchSomething()
            // ...
        }
    }
}
```

viewModelScope vs lifecycleScope

	viewModelScope	lifecycleScope
Owned by	A ViewModel	An Activity or Fragment
Cancelled when...	ViewModel is cleared (screen genuinely gone)	The Activity/Fragment is destroyed
Survives rotation?	Yes (the ViewModel itself survives)	No — cancelled on every rotation, like the Activity itself
Typical use	Business logic, data loading	UI-adjacent, one-off work tied to a specific screen instance

🔁 repeatOnLifecycle — Safely Collecting a Flow

Collecting a `StateFlow` naively inside `lifecycleScope.launch { }` keeps collecting even while the screen is fully backgrounded — wasted work, and potentially wasted battery/network. `repeatOnLifecycle` automatically starts and stops collection based on lifecycle state:

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
      viewModel.uiState.collect { state ->
        // Update Views here — this block only runs while STARTED or above
      }
    }
  }
}
```

`repeatOnLifecycle(Lifecycle.State.STARTED)` automatically cancels the collection when the Activity drops below `STARTED` (backgrounded — Course 1, Chapter 2's `onStop`) and restarts it when it returns to `STARTED` or above — no manual pause/resume bookkeeping required.

> ⚠ Compose Already Handles This For You
>
> Every Compose example so far in this course used `collectAsStateWithLifecycle()`, which does exactly this `repeatOnLifecycle` dance internally — it's why Chapter 2 through 5 never needed to think about this manually. `repeatOnLifecycle` matters directly when collecting a Flow inside a traditional Activity/Fragment (View system code, Course 1), which is genuinely the more manual and more error-prone path.

Dispatchers, In Practice

Room's `suspend` DAO functions (Chapter 3) and Retrofit's `suspend` API calls (Chapter 4) already run their actual I/O off the main thread automatically — the libraries handle their own dispatcher switching internally. Manually reaching for `withContext(Dispatchers.IO)` or `Dispatchers.Default` (Kotlin Intermediate Chapter 1) is mainly needed for work those libraries don't already cover:

```
suspend fun processLargeDataset(items: List<Item>): List<Result> {
    return withContext(Dispatchers.Default) {
        // Genuine CPU-bound work — sorting, filtering, heavy computation —
        // that would otherwise block whatever thread called this function
        items.map { expensiveTransform(it) }
    }
}
```

Already Handled Automatically

Room queries, Retrofit calls, `delay()` — any library function that's genuinely `suspend` and does I/O has typically already dispatched itself correctly. Calling these from `viewModelScope.launch { }` with no dispatcher argument is already correct.

Needs Manual withContext

Your own CPU-heavy pure-Kotlin computation, or interop with a blocking Java/Android API that isn't itself suspend-aware — these need an explicit `withContext(Dispatchers.Default)` or `Dispatchers.IO` to avoid blocking the caller's thread.

💻 Coding Challenges

Challenge 1: viewModelScope Cleanup

Write a ViewModel with a function startPolling() that launches a viewModelScope coroutine looping with delay(2000) and logging a message each time. In a comment, explain precisely when this loop stops, and why no explicit cancel() call is needed anywhere in the ViewModel itself.

Goal: Reinforce that viewModelScope cancellation is automatic and tied to ViewModel lifetime, not something to hand-manage.

→ Solution

Challenge 2: repeatOnLifecycle in an Activity

Write MainActivity.onCreate collecting a ViewModel's uiState StateFlow using lifecycleScope.launch { repeatOnLifecycle(...) { ... } }, updating a plain (non-Compose) TextView's text with each new state's value. Explain in a comment what would happen (in terms of wasted work) if repeatOnLifecycle were removed and the Flow collected directly instead.

Goal: Practice the manual Flow-collection pattern needed outside Compose.

→ Solution

Challenge 3: Choosing a Dispatcher

Write a suspend function calculatePrimesUpTo(limit: Int): List<Int> that computes prime numbers (reusing Kotlin Fundamentals Chapter 8's isPrime() extension function idea) up to limit, wrapped in the correct withContext dispatcher for CPU-bound work. Add a comment explaining why this needs an explicit dispatcher while a Room/Retrofit call in this chapter's other examples didn't.

Goal: Practice recognizing genuinely CPU-bound work and dispatching it correctly.

→ Solution

💡 A Recap Chapter, Made Concrete

Nothing in this chapter was a new coroutines concept — Kotlin Intermediate Chapters 1-2 already covered suspend functions, structured concurrency, and Dispatchers in full. What's new here is purely Android's specific scopes (viewModelScope, lifecycleScope) and the repeatOnLifecycle pattern, which only exist because Android screens have a lifecycle at all — a constraint that doesn't apply to Kotlin running anywhere else.

🎯 What's Next

Next chapter: **DataStore & Preferences** — replacing SharedPreferences, and the difference between proto and preferences DataStore.

💾 DataStore & Preferences

Room (Chapter 3) is for structured, queryable data — a list of tasks, notes, users. For small, simple settings (a theme preference, "has the user seen onboarding," a saved username), Jetpack DataStore is the modern tool — Flow-based, coroutine-native, and the direct replacement for the older SharedPreferences API.

Why Not SharedPreferences?

`SharedPreferences` was Android's original key-value storage API — still seen in older code, but with real problems DataStore was built specifically to fix:

No Built-In Observability

Reading a value is a synchronous, one-time snapshot — there's no native way to *observe* a value changing over time without a separate, manually-registered listener callback.

apply() vs commit() Confusion

`commit()` writes synchronously (risking a main-thread stall); `apply()` writes asynchronously but silently swallows any failure — neither is a genuinely safe default, and it's easy to pick the wrong one without realizing it.

🔑 Preferences DataStore — A Flow-Based Key-Value Store

```
val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name = "settings")

object PreferencesKeys {
    val USERNAME = stringPreferencesKey("username")
    val DARK_MODE = booleanPreferencesKey("dark_mode")
}


// Reading — as a Flow, exactly like a Room query
val username: Flow<String> = context.dataStore.data
    .map { prefs -> prefs[PreferencesKeys.USERNAME] ?: "Guest" }

// Writing — a suspend function
suspend fun saveUsername(context: Context, name: String) {
    context.dataStore.edit { prefs ->
        prefs[PreferencesKeys.USERNAME] = name
    }
}
```

`Context.dataStore by preferencesDataStore(...)` reuses Kotlin Intermediate Chapter 4's property delegation again — this creates a single DataStore instance tied to the app's Context, following the same one-shared-instance idea as Chapter 3's Room database and Chapter 4's Retrofit client. `.data` is itself a `Flow<Preferences>` — every read is inherently observable, with no separate listener API needed, and every write is a genuine `suspend` function rather than a fire-and-forget `apply()` call.

⚠ String Keys Are Still Just Strings

Preferences DataStore is a real improvement over SharedPreferences, but it's still fundamentally a loosely-typed key-value bag underneath — `stringPreferencesKey("username")` and a typo'd `stringPreferencesKey("usrname")` elsewhere compile fine and silently create two unrelated keys. This is exactly the problem Proto DataStore, next, solves properly.

📐 Proto DataStore — A Real Typed Schema

Proto DataStore stores a single, strongly-typed object (defined via a `.proto` schema file and Protocol Buffers code generation) instead of loose string-keyed values:

```
// user_prefs.proto
syntax = "proto3";

message UserPreferences {
  string username = 1;
  bool dark_mode = 2;
}
```

A Gradle plugin generates a real Kotlin class (`UserPreferences`) from this schema at build time — reading and writing become fully type-checked, with no string-keyed lookups anywhere, and no possibility of a typo'd key silently creating a second, unrelated value.

Preferences DataStore vs Proto DataStore

	Preferences DataStore	Proto DataStore
Schema	None — loose string keys	Defined in a .proto file
Type safety	Per-key, easy to typo	Fully typed generated class
Setup complexity	Low — no extra build step	Higher — needs the Proto Gradle plugin
Best for	A handful of simple, independent settings	A larger, structured preferences object

For most small apps, Preferences DataStore is the pragmatic default — Proto DataStore's extra setup pays off once there's a genuinely complex settings object worth modeling with a real schema, similar to reaching for Room over a flat file once data gets structured enough to need it.

🔌 Wiring DataStore Into the Existing Pattern

This slots into exactly the same repository/ViewModel/Hilt pattern already used for Room and Retrofit:

```
class SettingsRepository @Inject constructor(
    @ApplicationContext private val context: Context
) {
    val username: Flow<String> = context.dataStore.data
        .map { prefs -> prefs[PreferencesKeys.USERNAME] ?: "Guest" }

    suspend fun setUsername(name: String) {
        context.dataStore.edit { prefs -> prefs[PreferencesKeys.USERNAME] = name }
    }
}

@HiltViewModel
class SettingsViewModel @Inject constructor(
    private val repository: SettingsRepository
) : ViewModel() {
    val username: StateFlow<String> = repository.username
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), "Guest")

    fun updateUsername(name: String) {
        viewModelScope.launch { repository.setUsername(name) }
    }
}
```

Every piece here — `@Inject constructor` (Chapter 5), a `Flow` exposed as `StateFlow` via `.stateIn(...)` (Chapter 3), `viewModelScope.launch` (Chapter 6) — is a direct repeat of the architecture already established for Room. DataStore is simply another data source feeding the exact same pipeline.

DataStore vs Browser localStorage

	Browser localStorage	Android DataStore
API shape	localStorage.setItem(key, value) — synchronous	dataStore.edit { } — suspend, async by design
Reading reactively	No built-in mechanism (storage event is limited)	.data is a Flow — inherently observable
Value types	Strings only, manual JSON.parse/stringify for anything else	Typed keys (Preferences), or a real schema (Proto)

💻 Coding Challenges

Challenge 1: A Preferences DataStore Setting

Set up a preferencesDataStore named "app_settings", a booleanPreferencesKey for "notifications_enabled", and functions to read it as a Flow<Boolean> (defaulting to true if unset) and to write a new value via a suspend function.

Goal: Practice the basic Preferences DataStore read/write pattern.

→ Solution

Challenge 2: A Settings Repository with Hilt

Wrap Challenge 1's DataStore logic in a SettingsRepository with an @Inject constructor (taking @ApplicationContext Context), exposing notificationsEnabled as a Flow<Boolean> and a suspend setNotificationsEnabled(enabled: Boolean) function.

Goal: Practice wrapping DataStore access in a properly-injected repository, following Chapter 5's pattern.

→ Solution

Challenge 3: A Settings ViewModel

Write a @HiltViewModel SettingsViewModel exposing Challenge 2's notificationsEnabled as a StateFlow<Boolean> via stateIn, plus a toggleNotifications() function that reads the current value and writes its inverse via the repository.

Goal: Practice the complete DataStore → Repository → ViewModel chain, end to end.

→ Solution

💡 Course 2's Data Layer Is Now Complete

Room, Retrofit, and DataStore cover the three data sources a typical Android app actually needs — structured local data, remote data, and simple settings — all wired through the identical repository → Hilt-injected ViewModel → StateFlow → Compose pipeline established since Chapter 3. The final chapter pulls all of this together into one cohesive architecture.

🎯 What's Next

Next chapter — the final chapter of Course 2: **MVVM Architecture** — the repository pattern, clean architecture basics, and separating concerns across everything built so far.

MVVM Architecture

Every chapter since Chapter 2 has quietly been building toward the same architecture without naming it: MVVM. This final chapter names it explicitly, formalizes the repository pattern that's appeared in every data chapter, and shows why a ViewModel talking only to a repository interface — never directly to Room or Retrofit — is a deliberate design choice, not an accident.

🎭 MVVM, Named

Model

The data layer — Room entities and DAOs (Chapter 3), Retrofit API services (Chapter 4), DataStore (Chapter 7) — all unified behind repositories.

View

Compose composables (Chapter 1) — purely a function of whatever `UiState` they're given, with no direct knowledge of where that state came from.

ViewModel

Owns and exposes `UiState` as `StateFlow` (Chapter 2), reacting to user actions by calling into the Model layer — never touching a database or network call directly.

Nothing about this is new material — it's a name for the shape every chapter's code examples already had. Naming it matters because it turns "how should I structure this new screen?" into a question with a known, repeatable answer.

🗃️ The Repository Pattern, Properly

Earlier chapters' repositories each wrapped a single data source. A repository can just as easily combine *several*, presenting one unified API — this is the classic **offline-first** shape: check local data immediately, refresh from the network in the background, and let the local database remain the single source of truth the UI actually observes:

```
class TaskRepository @Inject constructor(
    private val dao: TaskDao,
    private val api: TaskApiService
) {
    // The UI only ever observes the LOCAL database — never the network directly
val tasks: Flow<List<Task>> = dao.getAllTasks()

    suspend fun refresh() {
        try {
            val remoteTasks = api.getTasks()
            remoteTasks.forEach { dao.insert(it) }   // writing to Room triggers tasks (the Flow) to re-emit
        } catch (e: IOException) {
            // Network failed — the UI keeps showing whatever's already cached locally
        }
    }
}
```

The UI never directly asks "give me the network version" or "give me the cached version" — it just observes `tasks`, and the repository decides how that data gets kept fresh. This is exactly why `refresh()`'s `catch` block can simply do nothing on failure: the already-displayed cached data from Room is a perfectly reasonable fallback, not an error state the UI needs to handle specially.

🚫 Why a ViewModel Shouldn't Know About Room or Retrofit Directly

Every ViewModel across this course took a repository as a constructor dependency, never a `TaskDao` or `ApiService` directly. This is **dependency inversion** — the ViewModel depends on an abstraction (what the repository exposes), not the concrete implementation behind it:

```
// A repository INTERFACE — the ViewModel only ever sees this
interface TaskRepository {
    val tasks: Flow<List<Task>>
    suspend fun refresh()
}

// The real implementation — Room + Retrofit specifics live ONLY here
class TaskRepositoryImpl @Inject constructor(
    private val dao: TaskDao,
    private val api: TaskApiService
) : TaskRepository {
    override val tasks = dao.getAllTasks()
    override suspend fun refresh() { /* ... */ }
}


// A Hilt module tells the framework which implementation to provide for the interface
@Module
@InstallIn(SingletonComponent::class)
abstract class RepositoryModule {
    @Binds
abstract fun bindTaskRepository(impl: TaskRepositoryImpl): TaskRepository
}
```

`@Binds` (a lighter alternative to `@Provides`, used specifically for "this interface, backed by this implementation" mappings) tells Hilt: whenever something asks for a `TaskRepository`, hand it a `TaskRepositoryImpl`. A ViewModel constructor asking for `TaskRepository` never knows or cares that Room and Retrofit exist underneath — swapping in a fake implementation for a test only requires changing this one binding, with the ViewModel's own code completely untouched.

⚠ Skipping the Interface Is Fine for a Small App

Every earlier chapter in this course used a concrete repository class directly (no interface) — that's a genuinely reasonable simplification for a small app or a learning example. The interface-plus-`@Binds` version shown here is what a larger, more seriously tested codebase adds once the flexibility (swappable implementations, easier fakes for tests) actually earns its extra ceremony — not a requirement for every app, every time.

📚 A Note on "Clean Architecture" Layers

Some codebases add a further **domain layer** between the UI and data layers — standalone "use case" classes (e.g. `GetTasksUseCase`, `RefreshTasksUseCase`) that a ViewModel calls instead of a repository directly, intended to hold business logic that doesn't belong in either the UI or the data layer. This course's examples skip that layer deliberately — for the scale of app built throughout these chapters, ViewModel-calls-repository-directly is a completely legitimate, widely-used architecture. A domain layer is worth reaching for once business logic (not just data fetching) grows complex enough to want its own dedicated, independently-testable home.

MVVM vs Frontend Architecture

	React (hooks + API layer)	Android MVVM
UI layer	Function components	Composables
State + logic owner	A custom hook, or a store	ViewModel
Data access abstraction	An API client module / service layer	Repository (interface + implementation)
Swapping real for fake data	Mock the API module	Swap the @Binds implementation

💻 Coding Challenges

Challenge 1: An Offline-First Repository

Write a NoteRepository combining Course 2's Note entity/NoteDao with a hypothetical NoteApiService, exposing notes as a Flow from the DAO and a suspend refresh() that fetches from the API and writes results into the DAO, with a try/catch that silently ignores network failure (falling back to cached data).

Goal: Practice the offline-first repository pattern combining two data sources.

→ Solution

Challenge 2: Interface + @Binds

Extract Challenge 1's NoteRepository into an interface NoteRepository plus an implementation NoteRepositoryImpl, and write a Hilt @Module with an abstract @Binds function mapping the interface to the implementation.

Goal: Practice the dependency-inversion setup, distinct from a plain @Inject-constructor class.

→ Solution

Challenge 3: The Full Stack, One More Time

Write a NoteViewModel depending only on the NoteRepository interface (not the impl), exposing a NotesUiState (notes, isLoading, errorMessage), calling refresh() on init, and a NotesScreen composable collecting and rendering that state — pulling together everything from Chapters 1 through 8 into one complete, working screen.

Goal: Assemble one final, complete example using every architectural piece from this course.

→ Solution

💡 Course 2 Complete — What This Sets Up

Architecture & Data closes here having covered Compose, ViewModel/StateFlow, Room, Retrofit, Hilt, coroutine scopes, DataStore, and now the MVVM pattern tying them together. Course 3 (Production & Publishing) shifts focus to shipping this kind of app for real — testing it, handling background work, securing it, optimizing performance, and getting it into the Play Store.

🎯 What's Next

Android Development — Architecture & Data (Course 2) is complete. Course 3 (Production & Publishing) begins with **Jetpack Compose Advanced** — custom layouts, animations, theming, and performance.