ANDROID DEVELOPMENT

Production & Publishing

Compose Advanced · Testing

Background Work · Notifications & Permissions

App Security · Performance

Publishing to Google Play · Firebase Integration

Philip Osztromok

Generated with Claude

Table of Contents

Android Development — Production & Publishing — 8 Chapters

CHAPTER	TITLE
Chapter 1	**Jetpack Compose Advanced**
Chapter 2	**Testing**
Chapter 3	**Background Work**
Chapter 4	**Notifications & Permissions**
Chapter 5	**App Security**
Chapter 6	**Performance & Optimisation**
Chapter 7	**Publishing to Google Play**
Chapter 8	**Firebase Integration**

🎬 Jetpack Compose Advanced

Course 2 built working screens with Compose; this chapter covers what turns a working screen into a polished, production-quality one — custom layout logic, real animations, app-wide theming beyond the defaults, and the recomposition pitfalls that quietly cause janky performance in larger apps.

📐 Custom Layouts

`Column` and `Row` (Course 2, Chapter 1) cover most needs, but the `Layout` composable gives full manual control over measuring and placing children — useful when no built-in layout produces the arrangement needed:

```
@Composable
fun StaggeredRow(modifier: Modifier = Modifier, content: @Composable () -> Unit) {
    Layout(content = content, modifier = modifier) { measurables, constraints ->
        val placeables = measurables.map { it.measure(constraints) }
        var yOffset = 0

        layout(constraints.maxWidth, placeables.sumOf { it.height }) {
            placeables.forEach { placeable ->
                placeable.placeRelative(x = 0, y = yOffset)
                yOffset += placeable.height
            }
        }
    }
}
```

Every child is **measured** first (asked how big it wants to be, given the available `constraints`), then **placed** at specific coordinates — this two-phase measure/place cycle is exactly what `Column` and `Row` do internally, just with their own arrangement logic instead of this example's simple vertical stacking. Reaching for a custom `Layout` is genuinely rare in practice — most real UI is built entirely from existing layout composables, so recognizing when it's actually necessary (rather than reaching for it prematurely) matters more than mastering every detail of the API.

🎞️ Animations

The simplest Compose animations are `animate*AsState` functions — they take a target value and automatically animate toward it whenever that target changes:

```
@Composable
fun ExpandableCard(expanded: Boolean) {
  val height by animateDpAsState(
    targetValue = if (expanded) 200.dp else 80.dp,
    label = "cardHeight"
  )
  Box(modifier = Modifier.height(height).fillMaxWidth())
}
```

`animateDpAsState` reuses the exact same `by` property delegation as `remember { mutableStateOf(...) }` (Course 2, Chapter 1) — `height` reads like a plain value, but every recomposition where `expanded` changes triggers a smooth transition rather than an instant jump.

```
@Composable
fun Notification(visible: Boolean) {
  AnimatedVisibility(visible = visible) {
    Text("You have a new message")
  }
}
```

`AnimatedVisibility` animates a composable's appearance/disappearance entirely (fade, slide, or a custom combination) rather than animating one property at a time — the right tool when something is conceptually "there or not there," not just changing size or color.

Compose Animation vs CSS

CSS	Compose
transition: height 0.3s ease;	animateDpAsState(targetValue = ...)
@keyframes / a mount-transition library	AnimatedVisibility
Coordinated multi-property transition	updateTransition (multiple synchronized animate*AsState-style values)

🎨 Theming Beyond the Defaults

Course 1 Chapter 8 covered XML themes (`themes.xml`, `colorPrimary`). Compose has its own, parallel theming system — `MaterialTheme`, typically wrapping the whole app:

```
private val AppColorScheme = lightColorScheme(
    primary = Color(0xFF3DDC84),
    onPrimary = Color(0xFF0F1117)
)

@Composable
fun MyAppTheme(content: @Composable () -> Unit) {
    MaterialTheme(
        colorScheme = AppColorScheme,
        typography = Typography(),
        content = content
    )
}
```

Any composable inside `MyAppTheme { }` can read `MaterialTheme.colorScheme.primary` instead of hardcoding a color — the exact same "define once, reference everywhere" idea as Course 1's XML theme attributes, just expressed as Kotlin objects instead of an XML resource. Dark mode support follows the same pattern too — a separate `darkColorScheme(...)`, chosen based on `isSystemInDarkTheme()`, replaces Course 1's `values-night` resource-qualifier folder mechanism with an explicit runtime check.

⚡ Performance — Avoiding Unnecessary Recomposition

Course 2 established that Compose recomposes only what reads changed state — but that guarantee depends on Compose being able to tell whether a composable's inputs actually changed, which isn't automatic for every type:

Stable Types Recompose Efficiently

A `data class` built entirely from `val`s (Kotlin Fundamentals Chapter 4) is considered **stable** — Compose can compare old vs new by value and skip recomposition if nothing changed. A regular class with `var`s, or a plain interface, may not offer this guarantee.

remember for Expensive Calculations

A costly computation inside a composable body re-runs on *every* recomposition unless wrapped in `remember(key) { ... }` — the same `remember` from Course 2, used here purely for caching a derived value, not holding mutable state.

```
@Composable
fun ExpensiveList(items: List<Item>) {
    val sortedItems = remember(items) { items.sortedBy { it.priority } }   // only re-sorts when "items" actually cha
nges

    LazyColumn {
        items(sortedItems, key = { it.id }) { item ->   // key() helps Compose track identity across list changes
            Text(text = item.name)
        }
    }
}
```

`remember(items) { ... }` takes `items` as a key — the sort only re-runs when `items` itself changes, not on every unrelated recomposition. `key = { it.id }` inside `LazyColumn`'s `items(...)` gives each row a stable identity across reorders/insertions/deletions — directly the same role React's `key` prop plays in a list, and DiffUtil's `areItemsTheSame` played back in Course 1's RecyclerView chapter.

⚠ derivedStateOf — For State Computed From Other State

`remember { derivedStateOf { someState > threshold } }` is a more specialized tool: it recomputes only when its own inputs genuinely change, and — critically — only triggers recomposition of composables reading *its* result when that result actually differs, not every time the underlying state updates. Useful for something like "is the list scrolled past item 5?" derived continuously from a fast-changing scroll position, where recomposing on every pixel of scroll would be wasteful.

💻 Coding Challenges

Challenge 1: An Animated Toggle

Write a composable ToggleCard(isActive: Boolean) that animates both its background color (animateColorAsState, between two colors) and its elevation/size (animateDpAsState) based on isActive, plus a Button that flips a remembered Boolean state to trigger it.

Goal: Practice combining two animate*AsState calls driven by the same state.

→ Solution

Challenge 2: A Custom MaterialTheme

Define a custom color scheme (at least primary and onPrimary) and wrap a small screen (a Text and a Button) in a MyAppTheme composable using it. Confirm the Button picks up the custom primary color automatically, the same way an unstyled Course 1 Button picked up colorPrimary from an XML theme.

Goal: Practice defining and applying a Compose-native theme.

→ Solution

Challenge 3: remember and key() in a List

Write a composable rendering a LazyColumn of a List<Product>, sorted by price using remember(products) { }, with a stable key = { it.id } on each item. Add a comment explaining what could go wrong (in terms of wasted work or incorrect item state) if either the remember or the key were removed.

Goal: Practice both performance techniques together and reason about why each matters.

→ Solution

💡 Profiling Beats Guessing

Performance techniques like `remember` and stable types are easy to over-apply "just in case." Android Studio's Layout Inspector and Compose-specific recomposition counts (enabled via a debug flag) show exactly which composables are recomposing and how often — worth reaching for before assuming where a real performance problem actually is, rather than optimizing blind.

🎯 What's Next

Next chapter: **Testing** — unit tests, instrumented tests, Espresso, and Compose UI testing.

🧪 Testing

Every architectural decision back in Course 2 Chapter 8 — a ViewModel depending on a repository interface, not Room or Retrofit directly — was set up specifically to make this chapter possible. This chapter covers unit tests (fast, no device needed), instrumented tests (need a real or virtual device), and Compose's own UI testing tools, which replace the older Espresso API for Compose screens.

⚡ Unit Tests — Fast, No Device Required

Unit tests live in `src/test/` and run directly on the local JVM — no emulator, no real device, seconds instead of minutes. They're the right tool for anything that's pure logic: a repository, a use case, a ViewModel, none of which genuinely need actual Android framework code to test.

```
// A fake implementing the SAME interface from Course 2, Chapter 8
class FakeNoteRepository : NoteRepository {
    private val _notes = MutableStateFlow<List<Note>>(emptyList())
    override val notes: Flow<List<Note>> = _notes.asStateFlow()

    var refreshCalled = false
private set
override suspend fun refresh() {
        refreshCalled = true
        _notes.value = listOf(Note(id = 1, title = "Test Note", content = "..."))
    }
}
```

This is exactly why Course 2 Chapter 8's interface-plus-`@Binds` pattern mattered: `NoteViewModel`'s constructor only ever demanded a `NoteRepository`, never `NoteRepositoryImpl` specifically — so a test can hand it this `FakeNoteRepository` instead, with zero Room, zero Retrofit, zero network or database access anywhere in the test.

```
@Test
fun `refresh updates uiState with notes`() = runTest {
    val fakeRepo = FakeNoteRepository()
    val viewModel = NoteViewModel(fakeRepo)

    viewModel.refresh()
    advanceUntilIdle()  // runs all pending coroutines to completion

    assertEquals(1, viewModel.uiState.value.notes.size)
    assertEquals("Test Note", viewModel.uiState.value.notes.first().title)
    assertTrue(fakeRepo.refreshCalled)
}
```

`runTest` (from `kotlinx-coroutines-test`) provides a special coroutine scope built for testing — it runs suspending code in a controlled, virtual-time environment, so `delay(2000)` anywhere in the code under test doesn't actually make the test wait 2 real seconds. Kotlin function names can be

arbitrary strings inside backticks (a Kotlin syntax feature, not test-specific) — a common convention for making test names read as full sentences.

📱 Instrumented Tests — Need a Real (or Virtual) Device

Instrumented tests live in `src/androidTest/` and run *on* an actual Android environment — an emulator or physical device — because they exercise real framework behavior a plain JVM can't fake: launching a real Activity, real Views, actual touch/click dispatch.

Unit Tests vs Instrumented Tests

	Unit Tests (src/test/)	Instrumented Tests (src/androidTest/)
Runs on	Local JVM	Emulator or real device
Speed	Fast — seconds	Slow — needs a booted device
Good for	ViewModels, repositories, pure logic	Real UI interaction, Activity lifecycle, actual rendering
Android framework access	None (or a limited fake via Robolectric)	Full, real

👆 Espresso — Testing the View System (Course 1)

Espresso drives and asserts against real Views, for apps (or screens) still using Course 1's XML/View system:

```
@Test
fun clickingButtonUpdatesText() {
    ActivityScenario.launch(MainActivity::class.java)

    onView(withId(R.id.submitButton)).perform(click())
    onView(withId(R.id.nameLabel)).check(matches(withText("Hello, Philip!")))
}
```

`onView(withId(...))` locates a real View by its resource ID (Course 1, Chapter 3), `.perform(click())` genuinely dispatches a click, and `.check(matches(...))` asserts on the resulting state — the same click-then-verify pattern behind every UI test, regardless of framework.

🖌️ Compose UI Testing — The Modern Equivalent

For Compose screens (Course 2 onward), a parallel API replaces Espresso's View-based lookups with node-based ones:

```
@get:Rule
val composeTestRule = createComposeRule()

@Test
fun clickingButtonUpdatesText() {
    composeTestRule.setContent {
        MyAppTheme { CounterScreen() }
    }

    composeTestRule.onNodeWithText("Increment").performClick()
    composeTestRule.onNodeWithText("Count: 1").assertIsDisplayed()
}
```

Espresso vs Compose Testing

	Espresso	Compose Testing
Finding an element	onView(withId(R.id.x))	onNodeWithText(...) / onNodeWithTag(...)
Action	.perform(click())	.performClick()
Assertion	.check(matches(withText(...)))	.assertTextEquals(...) / .assertIsDisplayed()
Setup	ActivityScenario.launch(...)	composeTestRule.setContent { ... }

`onNodeWithTag(...)` (paired with a `Modifier.testTag("...")` on the composable itself) is the more robust lookup when text alone isn't unique or reliable enough — the Compose-testing equivalent of Espresso's ID-based lookup, since Compose has no `R.id` resource system to key off of the way the View system does.

Android Testing vs JavaScript Testing

	JavaScript (Jest / Testing Library)	Android
Pure logic tests	Jest unit tests	JUnit unit tests (src/test/)
UI interaction tests	Testing Library (render, screen.getByText, fireEvent)	Compose UI tests (onNodeWithText, performClick)
Faking dependencies	jest.mock(...)	A fake implementation of an interface (this chapter's FakeNoteRepository)

💻 Coding Challenges

Challenge 1: A Fake Repository and ViewModel Unit Test

Write a FakeTaskRepository implementing Course 2's TaskRepository interface with a controllable in-memory list, then a unit test for TaskViewModel confirming that calling addTask("Buy milk") results in that task appearing in uiState.value.tasks, using runTest and advanceUntilIdle().

Goal: Practice the fake-dependency unit-testing pattern for a ViewModel.

→ Solution

Challenge 2: A Compose UI Test

Write a Compose UI test for Course 2's CounterScreen composable, using createComposeRule() to set its content, performing two clicks on the Increment button, and asserting the displayed count text reads "Count: 2".

Goal: Practice the full Compose testing setup: rule, setContent, node lookup, action, assertion.

→ Solution

Challenge 3: Choosing the Right Test Type

For each of the following, state whether a unit test or an instrumented/Compose UI test is appropriate, with a one-sentence reason: (a) NoteRepository's refresh() logic with a fake API, (b) confirming a Button's click actually navigates to a new screen, (c) a pure Kotlin function calculating tax on a price, (d) confirming a TextView's text updates after a ViewModel state change.

Goal: Practice the judgment call between test types, not just the syntax of either.

→ Solution

💡 Testability Was a Design Decision, Not an Afterthought

Every "why does this ViewModel take an interface instead of a concrete class" moment from Course 2 pays off directly in this chapter — a codebase built around concrete classes and singletons everywhere would make this chapter's fakes far harder, or impossible, to write cleanly. Good architecture and testability aren't separate concerns; they're the same decisions, viewed from two angles.

🎯 What's Next

Next chapter: **Background Work** — WorkManager, foreground services, and scheduling tasks.

🔧 Background Work

Every coroutine so far in this course (viewModelScope, lifecycleScope — Course 2, Chapter 6) is tied to a screen or ViewModel being alive. Some work genuinely needs to happen even after the app is closed — syncing data, uploading a file, sending a scheduled reminder. Android aggressively kills background processes to save battery, so this kind of work needs its own dedicated system: WorkManager.

🤔 Why This Needs a Dedicated System

Unlike a server process or a browser tab, an Android app can be killed by the OS at almost any time it's not visible — to reclaim memory, save battery, or because the user swiped it away. A plain `viewModelScope.launch { }` for something like "sync data every hour" would simply stop working the moment its owning ViewModel is gone, which could be seconds after the user leaves the screen. **WorkManager** exists specifically to survive all of that — process death, even a device reboot for persisted work — while still respecting the OS's own battery-saving constraints.

🛠️ Defining a Worker

```
class SyncWorker(
    context: Context,
    params: WorkerParameters
) : CoroutineWorker(context, params) {

    override suspend fun doWork(): Result {
        return try {
            // The actual background work — e.g. calling a repository's refresh()
            repository.refresh()
            Result.success()
        } catch (e: IOException) {
            Result.retry()   // WorkManager will reschedule this automatically
        }
    }
}
```

`doWork()` is a `suspend` function (Kotlin Intermediate Chapter 1) — the same coroutine fundamentals from every earlier chapter apply directly here. `Result.success()`, `Result.failure()`, and `Result.retry()` tell WorkManager what happened; `retry()` specifically triggers WorkManager's own backoff-and-reschedule logic, without any manual retry loop written by hand.

📤 Enqueueing Work

```
// A one-off task — runs once, as soon as constraints allow
val syncRequest = OneTimeWorkRequestBuilder<SyncWorker>().build()
WorkManager.getInstance(context).enqueue(syncRequest)

// A recurring task — repeats on an interval (minimum 15 minutes)
val periodicSync = PeriodicWorkRequestBuilder<SyncWorker>(1, TimeUnit.HOURS).build()
WorkManager.getInstance(context).enqueue(periodicSync)
```

OneTimeWorkRequest

Runs exactly once. The right choice for "upload this file now" or "sync after this specific user action" — a discrete, single task.

PeriodicWorkRequest

Repeats on an interval, with a 15-minute floor enforced by Android itself (battery protection, not a WorkManager limitation) — right for "check for updates every hour," wrong for anything needing tighter timing.

🔒 Constraints — Only Run When It Makes Sense

```
val constraints = Constraints.Builder()
    .setRequiredNetworkType(NetworkType.CONNECTED)
    .setRequiresCharging(true)
    .build()

val syncRequest = OneTimeWorkRequestBuilder<SyncWorker>()
    .setConstraints(constraints)
    .build()
```

WorkManager delays running the work until every constraint is satisfied — a sync task requiring a network connection simply won't attempt to run while the device is offline, rather than running and failing repeatedly. This declarative approach (state the requirements, let the system decide timing) mirrors this course's other declarative patterns — the nav graph (Course 1, Chapter 7) describing destinations rather than imperative navigation calls, Room's `@Query` describing what data is needed rather than how to fetch it.

🔔 Foreground Services — For Work the User Should See

WorkManager is for *deferrable* work — the user doesn't need to watch it happen in real time. Some work is the opposite: music playback, an active file download with a visible progress bar, a fitness-tracking session — work that should run immediately and stay visibly running via a persistent notification. That's a **foreground service**, a different tool for a genuinely different job:

WorkManager vs Foreground Service

	WorkManager	Foreground Service
User awareness	Invisible, deferrable	Visible, persistent notification required
Timing	"Whenever constraints allow"	Immediately, continuously, until stopped
Typical use	Sync, uploads, periodic maintenance	Music playback, navigation, active downloads
Survives process death?	Yes, automatically reschedules	No — the service itself is what's running

⚠ Choosing the Wrong One Is a Common Mistake

Reaching for a foreground service for something that's genuinely deferrable (like a periodic sync) forces an always-visible notification the user didn't ask for, and burns more battery than necessary. Reaching for WorkManager for something the user is actively watching (a download's progress bar) means the work could be delayed by constraints or battery optimization exactly when the user expects it to run immediately. Match the tool to whether the work is "the user is watching this right now" or "this can happen whenever."

WorkManager vs Node.js Background Jobs

	Node.js (BullMQ, Node Course 3)	Android WorkManager
Survives the process restarting?	Yes — jobs persist in Redis	Yes — WorkManager persists its own queue
Retry on failure	Configurable retry/backoff	Result.retry() + WorkManager's backoff policy
Scheduling	Delayed jobs, repeatable jobs (cron-like)	OneTimeWorkRequest / PeriodicWorkRequest
Constraints	Application-level (custom logic)	Built-in (network, charging, battery, storage)

💻 Coding Challenges

Challenge 1: A One-Time Sync Worker

Write a LogSyncWorker (CoroutineWorker) whose doWork() logs a message and returns Result.success(), and enqueue it as a OneTimeWorkRequest from a Button's click handler in a composable.

Goal: Practice the basic Worker + OneTimeWorkRequest + enqueue flow.

→ Solution

Challenge 2: A Constrained Periodic Worker

Write a PeriodicWorkRequest for Challenge 1's worker (or a new one) that repeats every 6 hours, with constraints requiring a connected network and NOT requiring the device to be charging. Add a comment explaining what happens if the network constraint isn't met when the 6-hour interval elapses.

Goal: Practice PeriodicWorkRequest with constraints, and reason about constraint-not-met behavior.

→ Solution

Challenge 3: WorkManager vs Foreground Service

For each scenario, state which tool is appropriate and why: (a) backing up notes to the cloud once a day, (b) playing a podcast episode while the app is backgrounded, (c) retrying a failed image upload with exponential backoff, (d) showing live GPS navigation directions.

Goal: Practice the judgment call between the two tools based on the chapter's "is the user watching this right now" heuristic.

→ Solution

💡 WorkManager Is the Bridge to "Real App" Concerns

Everything through Course 2 assumed the app was open and visible. This chapter is the first genuine step into what happens when it isn't — a distinction that barely exists for a typical web app (a closed tab just... stops) but is a constant, deliberate design consideration on mobile. The next few chapters (permissions, security, performance) continue in this same "production app" direction.

🎯 What's Next

Next chapter: **Notifications & Permissions** — notification channels, runtime permissions, and handling denials gracefully.

🔔 Notifications & Permissions

A background sync worker (last chapter) that finishes successfully but never tells the user anything is often incomplete — notifications are how Android apps communicate outside their own UI. Since Android 13, showing one requires a runtime permission, which is this chapter's second half: requesting permissions, and handling a user saying no gracefully rather than assuming yes.

📻 Notification Channels — Required Since Android 8

Every notification must belong to a **channel** — a named category the user can individually enable, disable, or customize (sound, vibration, priority) from system settings, independent of every other channel the app has:

```
val channel = NotificationChannel(
    "sync_updates",
    "Sync Updates",
    NotificationManager.IMPORTANCE_DEFAULT
).apply {
    description = "Notifications about background data sync"
}

val notificationManager = context.getSystemService(NotificationManager::class.java)
notificationManager.createNotificationChannel(channel)
```

Channels exist because a single blanket "allow notifications: yes/no" switch was too coarse — a user might want message notifications but not promotional ones from the same app. Creating a channel is idempotent (calling `createNotificationChannel` again with the same ID is a no-op if it already exists), so it's typically done once, early in the app's lifecycle — often inside the `Application` class from Course 2 Chapter 5's Hilt setup.

Building and Showing a Notification

```
val notification = NotificationCompat.Builder(context, "sync_updates")
    .setSmallIcon(R.drawable.ic_sync)
    .setContentTitle("Sync complete")
    .setContentText("Your notes are up to date")
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)
    .build()

if (ContextCompat.checkSelfPermission(context, Manifest.permission.POST_NOTIFICATIONS)
    == PackageManager.PERMISSION_GRANTED) {
    NotificationManagerCompat.from(context).notify(NOTIFICATION_ID, notification)
}
```

The permission check before `.notify(...)` isn't optional defensive code — without it, calling `notify` without the granted permission throws a `SecurityException` and crashes the app on Android 13+. This is the bridge into the chapter's second half: showing a notification and requesting permission to do so are two genuinely separate steps.

🔐 Runtime Permissions — Declared Isn't Enough

A "dangerous" permission (camera, location, notifications, and others) must be declared in `AndroidManifest.xml` *and* explicitly granted by the user at runtime — declaring it alone only makes the permission *requestable*, not automatically granted:

```
// AndroidManifest.xml — necessary but not sufficient on its own
<uses-permission android:name="android.permission.POST_NOTIFICATIONS" />


@Composable
fun NotificationPermissionRequester() {
    val launcher = rememberLauncherForActivityResult(
        contract = ActivityResultContracts.RequestPermission()
    ) { granted ->
        if (granted) {
            // Now safe to call NotificationManagerCompat.notify(...)
        } else {
            // Handled below — don't just silently give up
        }
    }

    Button(onClick = {
        launcher.launch(Manifest.permission.POST_NOTIFICATIONS)
    }) {
        Text("Enable Notifications")
    }
}
```

`rememberLauncherForActivityResult` is Compose's bridge to Android's activity-result APIs — `launcher.launch(...)` triggers the actual system permission dialog, and the lambda passed to `rememberLauncherForActivityResult` receives the user's choice as a plain `Boolean`, which then drives what the UI does next.

🚫 Handling Denial Gracefully

A denied permission isn't an error state to hide from — the app should keep working, just without that specific capability, and should explain *why* the permission matters before re-asking:

shouldShowRequestPermissionRationale

Returns `true` if the user denied once but hasn't permanently blocked it — the signal to show an explanatory message ("We use this to remind you about tasks") before asking again, rather than immediately re-prompting with no context.

Permanently Denied

After a user selects "Don't ask again" (or denies twice on some Android versions), the system stops showing the permission dialog entirely — the only path forward is directing the user to the app's system settings screen manually.

⚠ The App Must Function Without the Permission

A denied notification permission shouldn't mean the app is unusable — sync (last chapter) should still work, tasks should still save; the user simply won't be notified about it. Designing every permission-gated feature as an enhancement layered on top of a working core, rather than a hard requirement, is the difference between a graceful degradation and an app that feels broken the moment someone taps "Deny."

Android Permissions vs Browser Permissions

	Browser (Notification API, Geolocation API)	Android
Requesting	Notification.requestPermission() / navigator.geolocation	ActivityResultContracts.RequestPermission()
Result	"granted" / "denied" / "default"	A Boolean (granted or not)
Re-prompting after denial	Browser-controlled; often can't re-prompt at all from JS	Possible, guided by shouldShowRequestPermissionRationale
Permanent block	User must change it in browser site settings	User must change it in Android app settings

The underlying philosophy is identical to what's likely already familiar from the browser's own geolocation or notification prompts — the platform, not the app, controls whether and when to ask, and a denial is a real, permanent-feeling answer the app has to respect and design around.

💻 Coding Challenges

Challenge 1: A Notification Channel and a Notification

Create a notification channel named "task_reminders" with IMPORTANCE_DEFAULT, and a function showTaskReminder(context: Context, taskTitle: String) that builds and shows a notification using that channel, checking the POST_NOTIFICATIONS permission before calling notify().

Goal: Practice the full channel-then-notification setup.

→ Solution

Challenge 2: Requesting the Permission from Compose

Write a composable that checks whether POST_NOTIFICATIONS is already granted (using ContextCompat.checkSelfPermission), and if not, shows a Button that requests it via rememberLauncherForActivityResult, updating a remembered Boolean state based on the result.

Goal: Practice the check-then-request pattern for a runtime permission.

→ Solution

Challenge 3: Graceful Denial Handling

Extend Challenge 2: if the permission is denied, show explanatory text ("Notifications help you stay on top of tasks") only when shouldShowRequestPermissionRationale(activity, permission) returns true, and a different message ("Enable notifications in Settings") when it returns false after a denial. Confirm the rest of the screen (e.g. a placeholder Text saying "Tasks: 3") remains visible regardless of permission state.

Goal: Practice designing a feature that degrades gracefully rather than blocking the whole screen on a denied permission.

→ Solution

💡 Every Dangerous Permission Follows This Same Shape

Camera, location, contacts, microphone — every runtime-requestable permission uses the identical check → request → handle-denial-gracefully pattern from this chapter, just with a different permission string and a different fallback behavior. Learning this pattern once genuinely transfers to any permission a future feature might need.

🎯 What's Next

Next chapter: **App Security** — ProGuard/R8, certificate pinning, secure storage, and biometric authentication.

🔐 App Security

An installed APK sits on the user's device, fully extractable and inspectable — a fundamentally different threat model from a web server, which an attacker never gets to hold a copy of. This chapter covers four practical defenses: shrinking/obfuscating release code, pinning certificates against compromised CAs, storing secrets encrypted rather than plain, and biometric authentication for sensitive actions.

ProGuard/R8 — Shrinking and Obfuscating Release Builds

R8 (Android's modern successor to ProGuard, using the same rule syntax) strips unused code and renames classes/methods to short, meaningless names in release builds — smaller APKs, and a real (though not absolute) speed bump against casual reverse engineering:

```
// app/build.gradle.kts
buildTypes {
    release {
        isMinifyEnabled = true
        proguardFiles(getDefaultProguardFile("proguard-android-optimize.txt"), "proguard-rules.pro")
    }
}
```

R8's aggressive renaming breaks anything relying on reflection to find a class or method by its *original* name — Room, Retrofit/Moshi's JSON mapping, and Kotlin Intermediate Chapter 6's reflection all fall into this category. A `proguard-rules.pro` **keep rule** tells R8 to leave specific classes untouched:

```
// proguard-rules.pro — protects classes that reflection needs by real name
-keep class com.philip.myapp.data.** { *; }  # Entities, API response models — Moshi/Room need real field name
s
```

⚠ Test the Release Build, Not Just Debug

Debug builds have `isMinifyEnabled = false` by default, so a missing keep rule can go completely unnoticed until a release build crashes — often with a confusing `ClassNotFoundException` or a JSON field silently mapping to null, since Moshi/Room can no longer find the (renamed) class or property it expects. Genuinely testing a release build before shipping is what catches this class of bug.

📌 Certificate Pinning

The HTTPS/TLS course covered the certificate authority chain of trust — a client trusts any certificate signed by a CA in its trust store. **Certificate pinning** narrows that further: the app hardcodes exactly which certificate (or public key) it expects from a specific server, rejecting connections even to a certificate that's technically CA-valid but doesn't match the pin:

```
val certificatePinner = CertificatePinner.Builder()
    .add("api.myapp.com", "sha256/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=")
    .build()

val client = OkHttpClient.Builder()
    .certificatePinner(certificatePinner)
    .build()

val retrofit = Retrofit.Builder()
    .baseUrl("https://api.myapp.com/")
    .client(client)   // Retrofit (Course 2, Chapter 4) uses OkHttp underneath — this is where it's configured
    .addConverterFactory(MoshiConverterFactory.create())
    .build()
```

This defends against a specific, narrow threat: a compromised or coerced CA issuing a fraudulent-but-technically-valid certificate for the app's API domain, enabling a man-in-the-middle attack that a normal HTTPS connection wouldn't detect. Pinning is a deliberate trade-off, not a free upgrade — a legitimate certificate rotation on the server that isn't matched by an app update locks out every installed copy of the app until it's updated, which is why pinning is reserved for apps handling genuinely sensitive data, not applied by default everywhere.

🔒 Secure Storage — Beyond Plain DataStore

Course 2 Chapter 7's DataStore is unencrypted on disk — fine for a theme preference, wrong for an auth token or API key. The Jetpack Security library provides an encrypted alternative, backed by the Android Keystore system (hardware-backed key storage on supported devices):

```
val masterKey = MasterKey.Builder(context)
    .setKeyScheme(MasterKey.KeyScheme.AES256_GCM)
    .build()

val encryptedPrefs = EncryptedSharedPreferences.create(
    context,
    "secure_prefs",
    masterKey,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)

encryptedPrefs.edit().putString("auth_token", token).apply()
```

The API deliberately looks like ordinary `SharedPreferences` — the encryption is entirely handled underneath, with keys managed by the Android Keystore rather than stored alongside the encrypted data itself (which would defeat the purpose). This connects directly to the Auth course's coverage of session/token storage: on the web, an httpOnly cookie protects a token from JavaScript access; on Android, encrypted storage protects it from being read as plain text if the device's filesystem is ever accessed directly (a rooted device, physical access, a backup extraction).

👆 Biometric Authentication

`BiometricPrompt` gates access to a specific action or screen behind the device's fingerprint/face unlock — an additional local authentication factor, layered on top of (not replacing) whatever server-side auth the Auth course covered:

```
val biometricPrompt = BiometricPrompt(activity, executor,
    object : BiometricPrompt.AuthenticationCallback() {
        override fun onAuthenticationSucceeded(result: BiometricPrompt.AuthenticationResult) {
            // Proceed — e.g. reveal a sensitive screen, or decrypt something locally
        }
        override fun onAuthenticationFailed() {
            // Wrong fingerprint/face — the prompt itself already showed an error
        }
    })

val promptInfo = BiometricPrompt.PromptInfo.Builder()
    .setTitle("Unlock to view account")
    .setNegativeButtonText("Cancel")
    .build()

biometricPrompt.authenticate(promptInfo)
```

Mobile Security vs the Web Security Courses

Web Security Course Concept	Android Equivalent
CA chain of trust (HTTPS course)	Certificate pinning narrows trust further, per-app
httpOnly cookie protecting a session token (Auth course)	EncryptedSharedPreferences / Keystore-backed storage
MFA (Auth course, Chapter 6)	BiometricPrompt as a local, device-level factor
Minified/obfuscated JS as weak security-by-obscurity	R8 — similarly not a substitute for real access control, but raises the bar

💻 Coding Challenges

Challenge 1: A Keep Rule for a Reflection-Dependent Class

Given a data class ApiUser(val id: Int, val name: String) used as a Retrofit/Moshi response model, write the proguard-rules.pro keep rule that would protect it from R8's renaming, and explain in a comment specifically what would break (and how it would likely manifest) if this rule were missing in a release build.

Goal: Practice writing and reasoning about a ProGuard/R8 keep rule for a realistic scenario.

→ Solution

Challenge 2: Encrypted Token Storage

Write a TokenStorage class wrapping EncryptedSharedPreferences (using MasterKey with AES256_GCM), with functions saveToken(token: String) and getToken(): String? for storing/retrieving an auth token securely.

Goal: Practice the EncryptedSharedPreferences setup for a genuinely sensitive value.

→ Solution

Challenge 3: Biometric Gate for a Sensitive Screen

Write a function showBiometricPrompt(activity: FragmentActivity, onSuccess: () -> Unit) using BiometricPrompt to gate access, calling onSuccess() only on onAuthenticationSucceeded. Sketch (in a comment) how you'd use it in a composable to conditionally show an "Account Details" screen only after successful authentication.

Goal: Practice wiring BiometricPrompt behind a callback usable from the rest of the app.

→ Solution

💡 None of This Replaces Server-Side Security

Every technique in this chapter protects the client — the app installed on someone's device. It's not a substitute for anything covered in the security courses: proper authentication, authorization, input validation, and parameterized queries still have to be correct on the server, regardless of how well-defended the Android client is. A perfectly pinned, encrypted, obfuscated app talking to a server with a SQL injection vulnerability is still a broken system.

🎯 What's Next

Next chapter: **Performance & Optimisation** — profiling, memory leaks, ANR prevention, and baseline profiles.

📈 Performance & Optimisation

Chapter 1's tip box said "profile before optimizing" without saying how. This chapter covers the actual tool (Android Studio Profiler), the two failure modes that make a large fraction of Android performance complaints (memory leaks and ANRs — both largely preventable with patterns already used throughout this course), and baseline profiles, which optimize the part of an app's life every other technique here can't touch: the very first launch.

🔬 Profiling — Measure First

Android Studio's **Profiler** (View → Tool Windows → Profiler) attaches to a running app and shows CPU usage, memory allocation, and network activity in real time, against the actual device or emulator — not estimates, actual measurements:

CPU Profiler

Records a method trace showing exactly which functions consumed CPU time and for how long — the tool that turns "the app feels janky here" into "this specific function is the bottleneck."

Memory Profiler

Shows allocation over time and lets you capture a heap dump — a snapshot of every object currently in memory, which is exactly what's needed to actually confirm a suspected memory leak rather than just guess at one.

The recurring theme across every optimization technique in this chapter: measure first, then fix the specific thing the measurement points at. Applying `remember` everywhere (Chapter 1) or obsessing over a function that profiling shows takes 0.1% of frame time is wasted effort compared to fixing whatever the profiler actually flags as expensive.

Memory Leaks

A memory leak happens when something holds a reference to an object (most dangerously, an Activity or its Context) longer than that object's actual lifetime — the garbage collector can never reclaim it, because something still "needs" it, even though nothing legitimately does:

```
// A classic leak — a singleton holding an Activity Context indefinitely
object AnalyticsManager {
    private var context: Context? = null // DANGER: outlives any single Activity
fun init(context: Context) {
        this.context = context   // if this is an Activity Context, it can never be garbage collected
    }
}
```

This is exactly why Course 2 Chapter 5 was strict about `@ApplicationContext` in Hilt modules rather than an arbitrary `Context` — an `Application`-scoped context genuinely lives as long as the app does, so holding onto it long-term is safe; holding onto an Activity's Context the same way isn't. Other common sources: a listener registered on something long-lived (like a static object or a system service) and never unregistered, or a coroutine launched outside a properly-scoped `CoroutineScope` (Course 2, Chapter 6) that keeps running — and keeps its captured references alive — long after the screen using it is gone.

⚠ LeakCanary — Automated Leak Detection

LeakCanary (a debug-build-only library, added as a dependency) automatically detects leaked Activities/Fragments and shows a notification with the exact reference chain keeping them alive — turning "I suspect there's a leak somewhere" into a precise, actionable stack trace, without manually capturing and inspecting heap dumps by hand.

🚫 ANR — Application Not Responding

An ANR is triggered when the main thread is blocked for roughly 5 seconds — the system shows the user a dialog offering to close the app, which is about as bad an impression as an app can make. The root cause is almost always something blocking that should have been asynchronous:

```
// The wrong way — blocks the main thread
fun onButtonClick() {
    val data = repository.fetchDataBlocking()  // a synchronous, blocking network/database call — main thread f
rozen
}

// The right way — everything covered since Course 2 was building toward this
fun onButtonClick() {
    viewModelScope.launch {
        val data = repository.fetchData()  // suspend — Room/Retrofit already dispatch off the main thread
    }
}
```

Every architectural decision from Chapters 3-6 of Course 2 — `suspend` DAO/API functions, `viewModelScope.launch`, letting Room and Retrofit handle their own dispatching — exists specifically to make this class of bug structurally hard to write by accident. An ANR is rarely a mystery once traced: it's almost always a blocking call that skipped the coroutine pattern this entire course has been reinforcing.

🚀 Baseline Profiles — Optimizing First Launch

Android's runtime (ART) compiles app code just-in-time as it runs, on first launch — which means the very first time a user opens a newly-installed app, they experience the slowest possible version of it, before the JIT compiler has "warmed up." A **baseline profile** is a list of critical code paths (startup, common navigation flows) pre-compiled ahead of time and shipped inside the APK itself:

```
// A macrobenchmark test — generates the baseline profile
@Test
fun startup() = benchmarkRule.measureRepeated(
    packageName = "com.philip.myapp",
    metrics = listOf(StartupTimingMetric()),
    iterations = 5,
    startupMode = StartupMode.COLD
) {
    pressHome()
    startActivityAndWait()
}
```

Running this (via a separate macrobenchmark Gradle module) both measures actual cold-start time *and* generates the profile data that gets bundled into the release APK — subsequent installs benefit from ahead-of-time compilation for exactly the paths real users hit most (app launch, the first screen shown), without needing every possible code path pre-compiled.

Android Performance Metrics vs Web (Core Web Vitals)

Web (Core Web Vitals)	Android Equivalent
LCP (Largest Contentful Paint)	Cold-start time / Time to Full Display
INP (Interaction to Next Paint)	Jank / dropped frames during interaction
CLS (Cumulative Layout Shift)	No direct equivalent — Compose layout is more deterministic than browser reflow
Page freezing the browser tab	ANR (freezing the whole app)

💻 Coding Challenges

Challenge 1: Spot and Fix a Memory Leak

Given the AnalyticsManager singleton shown in this chapter's example (holding a raw Context), rewrite init(context: Context) to safely store only what's needed long-term (e.g. by taking context.applicationContext instead of the raw parameter). Explain in a comment why applicationContext specifically fixes the leak.

Goal: Practice recognizing and fixing the classic Context-leak pattern.

→ Solution

Challenge 2: Fix an ANR-Prone Function

Given a suspend fun loadUserProfile() that currently calls a blocking (non-suspend) legacy Java library function readFromDiskBlocking() directly, rewrite it to wrap that call in the correct withContext dispatcher (Course 2, Chapter 6) so it can't block the main thread when called from viewModelScope.launch.

Goal: Practice fixing a realistic ANR-prone pattern using the dispatcher knowledge from earlier in the course.

→ Solution

Challenge 3: Reading a Profiler Result

Given a hypothetical CPU Profiler trace showing 85% of a screen's load time spent inside a single function called parseAndSortLargeList(), and knowing this function currently re-runs on every recomposition with no memoization, explain (in a comment, referencing Chapter 1's tools specifically) what fix you'd apply and why profiling — not guessing — was what correctly identified this as the actual bottleneck.

Goal: Practice connecting a profiling result to the correct, already-learned fix (Chapter 1's remember).

→ Solution

💡 This Chapter Was Mostly a Payoff, Not New Rules

Memory leaks are largely prevented by the same @ApplicationContext-vs-Context discipline from Course 2 Chapter 5's Hilt setup. ANRs are largely prevented by the suspend/viewModelScope pattern from Course 2 Chapters 3-6. Getting this course's architecture right the first time is, in a real sense, already most of the performance work — this chapter mainly gave names and tools to problems the earlier patterns were already designed to avoid.

🎯 What's Next

Next chapter: **Publishing to Google Play** — signing, the app bundle format, Play Console, the store listing, and release tracks.

🚀 Publishing to Google Play

Everything so far has run on an emulator or a personally-connected device. This chapter covers the actual path to a real user's phone — signing a release build, the App Bundle format Google Play now requires, setting up a listing in Play Console, and release tracks, which let a build reach a small test audience before it reaches everyone.

🔑 App Signing — Every Release Build Needs a Key

An unsigned APK can't be installed on a real (non-debug) device at all — every release build must be signed with a private key that cryptographically proves subsequent updates genuinely come from the same developer. Android Studio's **Build → Generate Signed Bundle/APK** wizard creates a new keystore (a `.jks` file) on first use, or reuses an existing one:

Upload Key

The key used to sign the build actually uploaded to Play Console — kept by the developer, but if lost, Google Play's account recovery process can help re-establish a new one (this wasn't always true historically, which is why the next concept matters).

App Signing Key (Play App Signing)

Google Play itself holds the *real* signing key used for the final distributed APK, re-signing the uploaded bundle with it — this is now the default and strongly recommended, since Google securely manages the key that would otherwise be an unrecoverable single point of failure if lost.

⚠ Losing a Keystore (Before Play App Signing) Was Catastrophic

Before Play App Signing existed, losing the signing keystore meant permanently losing the ability to publish updates to an already-published app — a new app (a new listing, zero existing installs or reviews) was the only option. Play App Signing genuinely solves this, but it's worth understanding why signing keys were treated with such extreme care historically, and still deserve real care today (the upload key itself still matters).

📦 Android App Bundle (AAB) — The Modern Format

Google Play requires uploads in **AAB** format (`.aab`), not a traditional universal APK — a bundle contains everything for every device configuration, and Play itself generates and serves an optimized, smaller APK tailored to each specific installing device (screen density, CPU architecture, language) rather than shipping resources for every configuration to every user:

Universal APK vs App Bundle

	Universal APK	App Bundle (AAB)
Contains	Every resource, for every device config, in one file	Everything, but split and served per-device by Google Play
Download size for the user	Larger — includes unused configs	Smaller — only what that specific device needs
Where it's built	Locally, ready to install directly	Google Play's servers generate the final APK from the bundle
Google Play requirement	No longer accepted for new apps	Required

🖥️ Play Console — Setting Up the Listing

A new app in Play Console requires several pieces of information beyond just the build itself before it can go live:

- **Privacy policy URL** — required for essentially every app, hosted somewhere the developer controls.
- **Content rating questionnaire** — a series of questions (violence, user-generated content, etc.) that determines the age rating shown on the store listing.
- **Data safety form** — a declaration of exactly what data the app collects and how it's used/shared, shown to users before install — this maps directly onto whatever Course 2's networking and DataStore chapters actually collect and store, and must be accurate, not just filled in generically.

🖼️ Store Listing — What Users See Before Installing

Required Assets

A title (30 characters), a short description (80 characters), a full description, at least 2 screenshots, a feature graphic (1024×500), and an app icon — all directly visible on the store page before anyone taps Install.

App Store Optimization (ASO)

Choosing title/description wording that matches how people actually search, similar in spirit to SEO for a website (HTML course territory) — genuinely affects discoverability, not just aesthetics.

🛣️ Release Tracks — Reaching Users Gradually

A build doesn't have to go straight to every user — Play Console offers several tracks, each reaching a progressively wider audience:

Release Tracks, in Order of Reach

Track	Who Sees It
Internal testing	A small, manually-specified list (e.g. the dev team) — near-instant availability, no review delay
Closed testing (alpha)	A larger, invite-only group (an email list, a Google Group)
Open testing (beta)	Anyone who opts in via a public opt-in link — real, wider feedback before full release
Production	Everyone on Google Play — can itself be a staged rollout, e.g. 10% of users first

A staged production rollout (say, 10% → 50% → 100% over several days) lets a serious bug affect only a fraction of users before it's caught and halted — genuinely valuable given how much slower a store release is to "roll back" compared to redeploying a web server.

Publishing to Google Play vs Deploying a Web App

	Web (Node Course 3 deployment)	Google Play
Deploy speed	Minutes — push to a server/container	Hours to days — Google review process
Rollback	Redeploy the previous version immediately	Publish a new version and wait for review again
Gradual rollout	Feature flags, canary deploys, load balancer weighting	Staged rollout percentage, built into Play Console
User control over updates	None — the server just changes	User can delay/decline updating an installed app

That last row is the most consequential practical difference: unlike a web deploy where every user is instantly on the new version, an Android update only reaches users who actually update — which is why backward compatibility (a client from months ago still calling a live API) matters more here than it typically does for a tightly-coupled web frontend/backend pair.

💻 Coding Challenges

Challenge 1: Configuring Release Signing in Gradle

Write the signingConfigs and buildTypes blocks in app/build.gradle.kts for a release build referencing a keystore file, storePassword, keyAlias, and keyPassword (using placeholder values, sourced from local.properties or environment variables rather than hardcoded — explain why in a comment).

Goal: Practice the Gradle-side signing configuration, and the reasoning for keeping credentials out of source control.

→ Solution

Challenge 2: A Release Checklist

Write a checklist (as comments) of everything that must be true before a build can go live in production on Play Console, referencing specific requirements from this chapter (signing, AAB format, privacy policy, content rating, data safety form, store listing assets).

Goal: Practice recalling the full set of prerequisites, not just the technical build step.

→ Solution

Challenge 3: Choosing a Release Track

For each scenario, name the appropriate release track and why: (a) the very first build, wanting only the two-person dev team to try it, (b) a build ready for broader feedback from 500 opted-in volunteers, (c) a routine update to an already-live app, being cautious about a risky change.

Goal: Practice matching a release scenario to the correct track and rollout strategy.

→ Solution

💡 The Build Is Often the Easy Part

Everything through Chapter 6 makes a genuinely good app; this chapter is what actually gets it in front of real users. In practice, the non-technical pieces (privacy policy, content rating, store assets, waiting for review) often take longer than producing the signed AAB itself — worth planning for as real project time, not an afterthought squeezed in after "the code is done."

🎯 What's Next

Next chapter — the final chapter of this course: **Firebase Integration** — Crashlytics, Analytics, Remote Config, and FCM push notifications.

🔥 Firebase Integration

Once an app is live (last chapter), a whole category of question opens up that local testing can never answer: is it crashing on real devices? What are real users actually doing? Firebase covers exactly this — crash reporting, usage analytics, remote-controlled feature flags, and push notifications — closing out this course, and the Android Development series, with the tools that keep a published app healthy after release.

🔌 Connecting Firebase

A Firebase project is created at the Firebase console, an Android app is registered within it, and a generated `google-services.json` file is dropped into the app module — a Gradle plugin then reads it at build time to wire everything up:

```
// project-level build.gradle.kts
plugins {
    id("com.google.gms.google-services") version "4.4.0" apply false
}

// app/build.gradle.kts
plugins {
    id("com.google.gms.google-services")
}

dependencies {
    implementation(platform("com.google.firebase:firebase-bom:32.7.0"))
    implementation("com.google.firebase:firebase-crashlytics-ktx")
    implementation("com.google.firebase:firebase-analytics-ktx")
    implementation("com.google.firebase:firebase-config-ktx")
    implementation("com.google.firebase:firebase-messaging-ktx")
}
```

The Firebase BOM (Bill of Materials, via `platform(...)`) pins every Firebase library to mutually-compatible versions automatically — a similar dependency-management idea to how Gradle already resolves compatible versions for other libraries, just made explicit here since Firebase's own libraries need to agree with each other precisely.

💥 Crashlytics — Seeing Crashes You'll Never Reproduce Locally

Crashlytics automatically captures every fatal crash on a real user's device and reports it to the Firebase console — stack trace, device model, Android version, breadcrumbs leading up to it — none of which is visible from Android Studio's own Logcat once an app is out of your hands:

```
try {
    repository.refresh()
} catch (e: IOException) {
    FirebaseCrashlytics.getInstance().recordException(e)   // a NON-FATAL error, still reported
}
```

Fatal (app-crashing) exceptions are captured automatically with zero code — Crashlytics installs itself as an uncaught-exception handler. `recordException(e)` is for the other case: an error that was *handled* gracefully (per Chapter 4's approach to permission denial, or Course 2 Chapter 8's offline-first "silently keep showing cached data" pattern) but is still worth knowing about in aggregate — how often does this actually happen to real users?

📊 Analytics — What Users Actually Do

```
FirebaseAnalytics.getInstance(context).logEvent("task_created") {
    param("has_due_date", hasDueDate)
}
```

Logged events roll up into funnels and behavior reports in the Firebase console — how many users who install the app actually create a first task, how many come back the next day, which screens see the most engagement. This is the direct feedback loop the store listing chapter's ASO work feeds into: getting users to install is one problem; understanding what they do once they're in the app is a separate, ongoing one that only real usage data (not testing) can answer.

🎛️ Remote Config — Changing Behavior Without a New Release

Remote Config lets specific values be changed from the Firebase console and fetched by already-installed apps at runtime — no new build, no Play Console review, no waiting for users to update:

```
val remoteConfig = Firebase.remoteConfig
remoteConfig.setDefaultsAsync(mapOf("show_new_feature" to false))

remoteConfig.fetchAndActivate().addOnCompleteListener {
  val showNewFeature = remoteConfig.getBoolean("show_new_feature")
  // Use showNewFeature to conditionally render a composable, e.g. Course 2's UiState pattern
}
```

This is directly the same feature-flag idea Node.js Course 3's deployment chapter touched on for canary/gradual web rollouts — the difference is that on Android, given last chapter's slow, review-gated release cycle and users who don't always update immediately, Remote Config is often the *only* practical way to adjust behavior quickly across an already-installed user base, rather than one option among several fast-deploy strategies a web team might have.

📬 FCM — Push Notifications From a Server

Chapter 4 covered *local* notifications — triggered by code running inside the app itself. **FCM** (Firebase Cloud Messaging) is different: a message sent from a server (or the Firebase console directly) arrives on the device even when the app isn't running, and the app's own code decides how to display it:

```
class MyFirebaseMessagingService : FirebaseMessagingService() {
    override fun onMessageReceived(message: RemoteMessage) {
        val title = message.notification?.title ?: "New message"
val body = message.notification?.body ?: ""
// Builds and shows the notification using Chapter 4's exact same channel/Builder pattern
        showTaskReminder(applicationContext, "$title: $body")
    }

    override fun onNewToken(token: String) {
        // Send this device-specific token to your own server, so it can target this device later
    }
}
```

`onMessageReceived` ultimately calls the same notification-building code from Chapter 4 — FCM's job is only getting the message to the device reliably; actually displaying it as a notification (channel, `NotificationCompat.Builder`, permission check) is identical to everything already covered. `onNewToken` provides a unique identifier for this specific app install, which a backend needs on file to target that device individually later.

Local Notification (Chapter 4) vs FCM Push

	Local Notification	FCM Push Notification
Triggered by	Code running inside the app	A remote server, or the Firebase console
App needs to be running?	Yes — it's the one showing the notification	No — arrives even if the app isn't open
Display mechanism	NotificationCompat.Builder + channel	Same — FCM delivers the message, app code still builds the notification

💻 Coding Challenges

Challenge 1: Non-Fatal Error Reporting

Take Course 2 Chapter 8's NoteRepository.refresh() (the try/catch that silently falls back to cached data on IOException) and add a FirebaseCrashlytics.getInstance().recordException(e) call inside the catch block, explaining in a comment why this doesn't change the user-facing behavior at all but is still valuable.

Goal: Practice adding non-fatal error reporting to already-existing, correctly-handled error paths.

→ Solution

Challenge 2: An Analytics Event with Parameters

Add a FirebaseAnalytics logEvent call inside Course 2's TaskViewModel.addTask() function, logging an event named "task_added" with a parameter for the task title's character length (not the title itself — explain in a comment why logging the raw title would be a bad idea).

Goal: Practice logging a meaningful analytics event, and reasoning about what data is appropriate to send.

→ Solution

Challenge 3: A Remote Config Feature Flag

Write a function fun isNewDashboardEnabled(): Boolean using Firebase Remote Config with a default of false, fetched via fetchAndActivate(). Sketch (in a comment) how a composable would use this to conditionally show an old vs new dashboard screen, and explain what advantage this has over shipping the new dashboard directly in a release build.

Goal: Practice the Remote Config fetch pattern and articulate its real-world advantage over Chapter 7's release process.

→ Solution

💡 Course 3 Complete — And the Android Development Series

Compose animations and theming, testing, background work, permissions, security, performance, publishing, and now post-launch observability — Production & Publishing closes out a 24-chapter arc spanning Fundamentals, Architecture & Data, and this course, going from a first "Hello World" Activity to a signed, published, monitored app on real devices. Every architectural decision made early (Course 1's Activity lifecycle discipline, Course 2's testable repository interfaces, this course's dispatcher hygiene) is exactly what made the later, more production-focused chapters — testing, ANR prevention, non-fatal error reporting — straightforward rather than a fight against the codebase's own structure.

🎯 What's Next

Android Development — Production & Publishing (Course 3) is complete, and with it, the full Android Development series (Fundamentals → Architecture & Data → Production & Publishing).