

ANDROID COURSE

Android Development with Compose: Production & Publishing

8 Chapters

Philip Osztromok

Generated with Claude

ANDROID DEVELOPMENT WITH COMPOSE: PRODUCTION & PUBLISHING

Contents

Chapter 1	Advanced Compose	2
Chapter 2	Testing	11
Chapter 3	Background Work	21
Chapter 4	Notifications & Permissions	29
Chapter 5	App Security	37
Chapter 6	Performance & Optimisation	46
Chapter 7	Publishing to Google Play	54
Chapter 8	Firebase	60

Android Development with Compose: Production & Publishing

Course 6 · Chapter 1 · Beyond the Basics

The notes app from Android Development with Compose: Architecture & Data works, but a production app is judged on how it *feels*: smooth, responsive, consistent. This chapter covers the Compose tools for that: building your own modifiers and layouts, animating changes, and keeping recomposition cheap.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked in the docs (three searches, summaries only): Layout: parents pass constraints down and children report a size within them; the `layout` modifier gives you a measurable and the constraints, `measurable.measure(constraints)` makes a placeable, and `layout(width, height) { ... }` sets the size and holds the placement; the `Layout` composable gives `measurables` and `constraints`; `placeRelative` follows the layout direction and `place` does not; placing is only allowed after measuring, inside the layout pass; and for constraint-dependent measuring the docs prefer `Modifier.layout` or a `LayoutModifierNode` over `SubcomposeLayout` or `BoxWithConstraints`. **Animation:** the four tools and the "Choose an animation API" guidance below, including that `animateContentSize` goes before the size modifiers. **Performance:** defer state reads to layout or draw with lambda modifiers such as `Modifier.offset { ... }`; never write state that was already read in the same composition; give lazy layouts stable keys and content types; flat data classes with `val` primitive or String properties are inferred stable so `@Immutable` is redundant; skipping is not free; keep a list's instance when its content has not changed; measure with benchmarks, not recomposition counts alone; and, from one result, **a newer `computedStateOf` that the docs now recommend over `derivedStateOf` for most cases** (introduced in a Compose 1.13 release; I did not confirm its signature, so the code below uses the long-established `derivedStateOf`). **General knowledge, not checked:** the custom-modifier extension and `Layout` code as written, the `ExpandableNote` and scroll-to-top composables, the theme extension, "strong skipping" as a compiler default, and the Layout Inspector remarks. Theming was *not* searched in this chapter; Android Development Fundamentals with Compose 8 has the checked material. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

Custom Modifiers

(docs) The `layout` modifier lets you change how one element is measured and placed. You measure the wrapped element, report a size, and place it:

```
import androidx.compose.ui.Modifier
import androidx.compose.ui.layout.layout
import androidx.compose.ui.unit.Dp

// A custom modifier: add space above the first line of content
fun Modifier.paddingFromTop(top: Dp) = layout { measurable, constraints ->
    val placeable = measurable.measure(constraints)             // 1. measure the wrapp
    val extra = top.roundToPx()
    layout(placeable.width, placeable.height + extra) {          // 2. report a size
        placeable.placeRelative(x = 0, y = extra)                // 3. place it
    }
}
```

The three steps are always the same: **measure** the child with the incoming constraints, **report** your own size with `layout(width, height)`, then **place** the child in the placement block. *(docs)* Placing is only allowed in that block, after measuring. *(general knowledge)* A modifier that only combines existing modifiers can be an ordinary extension function that returns `this.padding(...).background(...)` without any measuring.

A Custom Layout

```
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.layout.Layout

// A custom layout: children stacked vertically, like a simple Column
@Composable
fun SimpleColumn(modifier: Modifier = Modifier, content: @Composable () -> Unit) {
    Layout(content = content, modifier = modifier) { measurables, constraints ->
        val placeables = measurables.map { it.measure(constraints) }

        val width = (placeables.maxOfOrNull { it.width } ?: 0)
            .coerceIn(constraints.minWidth, constraints.maxWidth)
        val height = placeables.sumOf { it.height }
            .coerceIn(constraints.minHeight, constraints.maxHeight)

        layout(width, height) {
            var y = 0
            placeables.forEach { placeable ->
```

```
                placeable.placeRelative(x = 0, y = y)
                y += placeable.height
            }
        }
    }
}
```

- *(docs)* `Layout` gives you `measurables` (the children) and the incoming `constraints`. Each child is measured once, then placed.
- *(docs)* `placeRelative` flips for right-to-left languages; `place` does not. Prefer `placeRelative` unless you want a fixed side.
- *(general knowledge)* The `coerceIn` calls keep the reported size inside the constraints; a layout that reports an impossible size is a bug.
- *(docs)* When measurement depends on constraints, the performance guidance prefers `Modifier.layout` or a `LayoutModifierNode` over `SubcomposeLayout` or `BoxWithConstraints`, which do more work.

You rarely need your own layout: `Column`, `Row`, `Box` and `LazyColumn` cover most screens. Write one when no built-in arrangement matches, for example a flow of tags that wraps, or a layout that overlaps children in an unusual way.

Animation: Choosing the Right Tool

YOU WANT TO ANIMATE…	USE *(DOCS)*
A composable appearing or disappearing	`AnimatedVisibility` (enter and exit transitions)
A change of size	`Modifier.animateContentSize()`, placed before size modifiers
One independent value (alpha, offset, padding, colour)	An `animate*AsState` function: you give the target, it animates from the current value, and adjusts if the target changes mid-way
Swapping between two layouts	`Crossfade`; for more than a fade, `AnimatedContent`

```
// One value, animated: you give the target and the state follows it
val alpha by animateFloatAsState(
    targetValue = if (enabled) 1f else 0.4f,
```

```
        label = "alpha"
    )
    Text("Sync", modifier = Modifier.alpha(alpha))
```

(docs) One difference to know: animating alpha with `animateFloatAsState` can leave an invisible item in the accessibility tree, whereas `AnimatedVisibility` eventually removes it from the composition. For something that should really go away, use `AnimatedVisibility`.

Putting It In the Notes App

An expandable note card, using `animateContentSize`:

```
import androidx.compose.animation.animateContentSize
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Card
import androidx.compose.material3.Text
import androidx.compose.runtime.*
import androidx.compose.runtime.saveable.rememberSaveable
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp

@Composable
fun ExpandableNote(text: String, modifier: Modifier = Modifier) {
    var expanded by rememberSaveable { mutableStateOf(false) }

    Card(
        modifier = modifier
            .animateContentSize()                // before the size modifiers, per the
            .fillMaxWidth()
            .clickable { expanded = !expanded }
    ) {
        Text(
            text = text,
            maxLines = if (expanded) Int.MAX_VALUE else 2,
            overflow = TextOverflow.Ellipsis,
            modifier = Modifier.padding(16.dp)
        )
```

```
    }
}
```

A scroll-to-top button, using `AnimatedVisibility` and `derivedStateOf`, in the shape the docs use as their own example, together with stable keys and content types for the lazy list:

```
import androidx.compose.animation.AnimatedVisibility
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.lazy.rememberLazyListState
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.KeyboardArrowUp
import androidx.compose.material3.FloatingActionButton
import androidx.compose.material3.Icon
import androidx.compose.runtime.*
import kotlinx.coroutines.launch

@Composable
fun NotesList(notes: List<Note>) {
    val listState = rememberLazyListState()
    val scope = rememberCoroutineScope()

    // Recomputes only when the answer changes, not on every scroll pixel
    val showScrollToTop by remember { derivedStateOf { listState.firstVisibleItemIndex >

    Box {
        LazyColumn(state = listState) {
            items(notes, key = { it.id }, contentType = { "note" }) { note ->
                ExpandableNote(text = note.text)
            }
        }
        AnimatedVisibility(visible = showScrollToTop, modifier = Modifier.align(Alignmen
            FloatingActionButton(onClick = { scope.launch { listState.animateScrollToIte
                Icon(Icons.Filled.KeyboardArrowUp, contentDescription = "Scroll to top")
            }
        }
    }
}
```

- `derivedStateOf` *(docs)* turns a fast-changing value (the scroll position) into a value that changes rarely (is the button needed?), so the composable only recomposes when the

answer changes. *(docs)* The newer `computedStateOf` is now recommended for most cases, with `derivedStateOf` kept for heavy calculations read several times per pass. Check your Compose version and the current docs. A common mistake the docs warn about is passing the state you read as a `remember` key, which recreates the block on every change.

- `key` **and** `contentType` *(docs)* in the lazy list let it skip work and reuse slots. Here the note's id is the key (it is unique and stable).

Keeping Recomposition Cheap

```
// Reads state during composition: the whole composable recomposes on every change
Box(Modifier.offset(x = scrollOffset.dp))

// Reads state in the layout phase: only layout reruns, composition is skipped
Box(Modifier.offset { IntOffset(x = scrollOffset, y = 0) })
```

- **Defer reads.** *(docs)* Compose works in phases: composition, layout, draw. A lambda modifier such as `Modifier.offset { ... }` reads the state later, in the layout phase, so a value that changes every frame no longer re-runs composition.
- **No backwards writes.** *(docs)* Never write to state that has already been read in the same composable's composition; it causes extra recompositions.
- **Stability.** *(docs)* Compose can skip a composable whose inputs have not changed.

```
// Stable by inference (docs): all val properties, primitive or String types. @Immutable
data class Note(val id: Long, val text: String, val createdAt: Long)

// A list parameter: keep the same list instance when the content has not changed
@Composable
fun NotesList(notes: List<Note>) { /* ... */ }
```

- *(docs)* Flat data classes with `val` properties of primitive or String types are inferred stable, so adding `@Immutable` is redundant. `@Stable` or `@Immutable` is worth it only in narrow cases, such as classes from non-Compose modules.
- *(docs)* Skipping is not free: each parameter of a skippable composable is stored as if remembered, so do not try to force skipping everywhere.

- *(docs)* For collections, keep the list instance the same when its content has not changed; reach for an immutable-list type only if a benchmark justifies it.
- *(general knowledge, not checked)* Recent Compose compilers skip more composables by default ("strong skipping"); the Compose compiler's stability reports show what the compiler decided for your classes.
- *(docs)* **Measure.** Do not judge performance by recomposition counts alone; use benchmarks (Chapter 6 of this course returns to measuring).

Extending the Theme

```
import androidx.compose.runtime.staticCompositionLocalOf
import androidx.compose.runtime.CompositionLocalProvider
import androidx.compose.ui.unit.Dp
import androidx.compose.ui.unit.dp

// Extending the theme with your own tokens
data class Spacing(val small: Dp = 8.dp, val medium: Dp = 16.dp, val large: Dp = 24.dp)

val LocalSpacing = staticCompositionLocalOf { Spacing() }

@Composable
fun NotesTheme(content: @Composable () -> Unit) {
    CompositionLocalProvider(LocalSpacing provides Spacing()) {
        MaterialTheme(content = content)       // colour scheme, typography, shapes as i
    }
}

// Use it: Modifier.padding(LocalSpacing.current.medium)
```

(general knowledge, not checked in this chapter) A theme is more than colours: a `CompositionLocal` lets you add your own tokens (spacing, elevations) that any composable below can read, without passing them as parameters. Colour schemes, dark theme and dynamic colour are covered, from checked sources, in Android Development Fundamentals with Compose 8.

> **COMPARED WITH THE CLASSIC UI COURSE**

Android Development: Production & Publishing 1 teaches Jetpack Compose Advanced: custom layouts, animations, theming and performance. The topics are the same; this edition adds the docs' current performance guidance (deferred reads, stability inference, benchmarks over recomposition counts) and the animation-API decision guide.

Try It Yourself

EXERCISE 1

Write `paddingFromTop` and `SimpleColumn`, then preview `SimpleColumn` with three text children, once alone and once inside a `Box` with a narrow width. Report how it behaves when the constraints are tight, and what happens if you remove the `coerceIn` calls.

View solution

EXERCISE 2

Add `ExpandableNote` and the scroll-to-top button to the notes app. Test with at least 40 notes. Then check your Compose version, look up `computedStateOf` in the current docs, and say whether it applies to you.

View solution

EXERCISE 3

Find one value in your app that changes often (for example an animated offset) and move its read into a lambda modifier. Explain why this skips composition, then say which tools you would use to confirm the change helped, and why recomposition counts alone are not enough.

View solution

NOT YET VERIFIED

Checked in the docs (three searches, summaries only): constraints, `measure`, `layout(width, height)` and placement; `Layout`'s `measurables`; `placeRelative` against `place`; placing only in the layout pass; the preference for `Modifier.layout` or `LayoutModifierNode` over `SubcomposeLayout` and `BoxWithConstraints`; the animation tools, the choice guidance, the alpha and accessibility-tree

note and `animateContentSize` placement; deferred state reads; backwards writes; stable keys and content types; stability inference and `@Immutable` redundancy; skipping not being free; list instances; measuring over counting; the `computedStateOf` recommendation.

General knowledge, not checked: the code as written (including every import and the `roundToPx` call); the custom-modifier-by-composition remark; strong skipping as a compiler default; the Compose compiler stability reports; the `CompositionLocal` theme extension and the idea that a theme is more than colours.

Not verified: the `computedStateOf` signature and the Compose version that introduced it; every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.

Sources: Custom layouts (developer.android.com), Create custom modifiers (developer.android.com), Quick guide to Animations in Compose (developer.android.com), Choose an animation API (developer.android.com), Follow best practices (developer.android.com), Fix stability issues (developer.android.com), Modifier breakdown by phase (developer.android.com).

COMING UP

Chapter 2 is Testing, which the previous course kept promising: the DAO, ViewModel and serialization tests done properly, Compose UI tests, and a Room migration test.

Android Development with Compose: Production & Publishing

Course 6 · Chapter 2 · Proving It Works

Android Development with Compose: Architecture & Data kept pointing here: "testing gets its own treatment". The architecture you built was designed for it. Because the screen is stateless, the ViewModel takes its repository in its constructor and the data layer has fakes, most of your app can be tested without running the whole thing.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked in the docs (two searches, summaries only): Compose UI tests: the setup needs the `ui-test-junit4` dependency and a debug manifest dependency for `createComposeRule()`; `createComposeRule()` when no specific activity is needed, `createAndroidComposeRule` when you need the activity; the sample pattern `onNodeWithText(...).performClick()` then `.assertIsDisplayed()`; the tests wait for the UI to be idle before acting; they are instrumented tests under `src/androidTest`; the Compose rules use a `StandardTestDispatcher` by default; and **combining `createComposeRule` with `runTest` gives two separate clocks**, so advancing time in one does not advance the other, with `runComposeUiTest` (in the v2 testing APIs) as the fix. **Room migration tests:** exported schema files should be kept in version control; the schema location is set through Room's Gradle plugin; `MigrationTestHelper` creates a database at an old version, you insert data with raw SQL, then `runMigrationsAndValidate` migrates and validates the schema, but you must check the data yourself; auto-migrations need `exportSchema = true`; and there is an older `room-testing` artifact and a newer `room3-testing` one, so check which your project uses. **From earlier chapters:** `Dispatchers.setMain` with a test dispatcher, and fakes or mocks for a ViewModel's repository. **Not found in the searches (general knowledge):** the split between local and instrumented tests, `backgroundScope`, `performTextInput`, `testTag`, the Gradle plugin's exact id, and the migration code. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

What Kind of Test Where

TEST	WHERE IT RUNS	FOLDER	SPEED
JSON parsing, mappers, plain logic	Your computer (local)	`src/test`	Seconds
ViewModel with fakes	Local	`src/test`	Seconds
Room DAO (in-memory database)	Device or emulator (instrumented)	`src/androidTest`	Slower

TEST	WHERE IT RUNS	FOLDER	SPEED
Room migration	Instrumented	`src/androidTest`	Slower
Compose UI *(docs: instrumented)*	Instrumented	`src/androidTest`	Slower

(general knowledge) Local tests need no Android at all, so you can run them dozens of times a minute; instrumented tests need an emulator or phone. Put as much as you can in the first group, which is exactly what the earlier chapters' structure allows.

```
// Module build.gradle.kts (versions are placeholders; the Compose test libraries follow
dependencies {
    // Local tests (src/test): run on your computer
    testImplementation("junit:junit:<junit-version>")
    testImplementation("org.jetbrains.kotlinx:kotlinx-coroutines-test:<coroutines-versio

    // Instrumented tests (src/androidTest): run on a device or emulator
    androidTestImplementation("androidx.compose.ui:ui-test-junit4")
    androidTestImplementation("androidx.test.ext:junit:<androidx-test-version>")
    androidTestImplementation("androidx.room:room-testing:<room-version>")     // Room 3

    // Needed by createComposeRule(): the test manifest, in debug builds only
    debugImplementation("androidx.compose.ui:ui-test-manifest")
}
```

Local Tests: the ViewModel

(docs, from the coroutines chapter's sources) A ViewModel's `viewModelScope` uses `Dispatchers.Main`, which does not exist in a plain local test. The fix the docs give is to replace Main with `Dispatchers.setMain` and a test dispatcher. Wrapping that in a JUnit rule means each test class needs only one line:

```
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.ExperimentalCoroutinesApi
import kotlinx.coroutines.test.TestDispatcher
import kotlinx.coroutines.test.UnconfinedTestDispatcher
import kotlinx.coroutines.test.resetMain
import kotlinx.coroutines.test.setMain
import org.junit.rules.TestWatcher
```

```
import org.junit.runner.Description

// Replaces Dispatchers.Main for every test that uses it (docs: Dispatchers.setMain)
@OptIn(ExperimentalCoroutinesApi::class)
class MainDispatcherRule(
    private val dispatcher: TestDispatcher = UnconfinedTestDispatcher()
) : TestWatcher() {
    override fun starting(description: Description) = Dispatchers.setMain(dispatcher)
    override fun finished(description: Description) = Dispatchers.resetMain()
}
```

```
// FakeNotesRepository: let a test choose what refresh() returns
class FakeNotesRepository : NotesRepository {
    var refreshResult: RefreshResult = RefreshResult.Success

    // ...the notes flow, addNote and deleteNote from Chapter 5...

    override suspend fun refresh(): RefreshResult = refreshResult
}
```

```
// app/src/test/.../NotesViewModelTest.kt  (a local test: fast, no emulator)
class NotesViewModelTest {

    @get:Rule
    val mainDispatcherRule = MainDispatcherRule()

    private fun viewModel(repository: FakeNotesRepository = FakeNotesRepository()) =
        NotesViewModel(repository, FakeSettingsRepository())

    @Test
    fun emptyNoteShowsMessage() = runTest {
        val vm = viewModel()
        backgroundScope.launch(UnconfinedTestDispatcher(testScheduler)) { vm.uiState.col

        vm.addNote("   ")

        assertEquals("A note cannot be empty", vm.uiState.value.userMessage)
    }

    @Test
    fun failedRefreshShowsMessage() = runTest {
        val repository = FakeNotesRepository().apply { refreshResult = RefreshResult.NoC
```

```
        val vm = viewModel(repository)
        backgroundScope.launch(UnconfinedTestDispatcher(testScheduler)) { vm.uiState.col

        vm.refresh()

        assertEquals("No connection", vm.uiState.value.userMessage)
    }
}
```

- **Fakes over mocks, here.** *(docs)* The guides say tests can pass fakes or mocks. A fake is a small working stand-in (Chapter 5's `FakeNotesRepository`); *(general knowledge)* it keeps tests readable and survives refactoring better than a mocking library's setup.
- **Why the collector.** The ViewModel's state uses `WhileSubscribed` (Chapter 3), so without a collector the upstream does not run and `uiState.value` would not change. `backgroundScope` *(general knowledge)* is a scope from `runTest` that is cancelled when the test ends.
- `runTest` runs coroutine code with a virtual clock, so `delay` costs no real time *(general knowledge)*.

Instrumented Tests: the DAO

```
// app/src/androidTest/.../NoteDaoTest.kt  (an instrumented test: needs a device or emul
@RunWith(AndroidJUnit4::class)
class NoteDaoTest {
    private lateinit var db: AppDatabase
    private lateinit var dao: NoteDao

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

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

    @Test
    fun insertedNoteIsObserved() = runTest {
```

```
        dao.insert(NoteEntity(text = "Buy milk"))
        assertEquals(listOf("Buy milk"), dao.observeNotes().first().map { it.text })
    }
}
```

(general knowledge) An in-memory database vanishes when closed, so every test starts clean. This is the Chapter 4 test again, now with the runner annotation it needs. *(docs)* It lives in `src/androidTest` and needs a device or emulator.

Compose UI Tests

Chapter 1 of Android Development with Compose: Architecture & Data made `NotesScreen` stateless: it takes a state and callbacks. That is what makes it testable: no ViewModel, no database; you hand it a state and look at the result.

```
// app/src/androidTest/.../NotesScreenTest.kt  (a Compose UI test)
class NotesScreenTest {

    @get:Rule
    val composeTestRule = createComposeRule()

    private fun show(state: NotesUiState, onAddNote: (String) -> Unit = {}) {
        composeTestRule.setContent {
            NotesScreen(
                state = state,
                onAddNote = onAddNote,
                onDeleteNote = {}, onRefresh = {}, onNewestFirstChange = {}, onMessageSh
            )
        }
    }

    @Test
    fun showsTheNotes() {
        show(NotesUiState(notes = listOf(Note(1, "Buy milk", 0L))))

        composeTestRule.onNodeWithText("Buy milk").assertIsDisplayed()
    }

    @Test
    fun addButtonSendsTheTypedText() {
        var added: String? = null
```

```
        show(NotesUiState(), onAddNote = { added = it })

        composeTestRule.onNodeWithTag("noteInput").performTextInput("Call Ada")
        composeTestRule.onNodeWithText("Add").performClick()

        assertEquals("Call Ada", added)
    }
}

// In the screen: OutlinedTextField(..., modifier = Modifier.testTag("noteInput"))
```

- *(docs)* `createComposeRule()` is the one to use when you do not need a specific activity; `createAndroidComposeRule` gives access to the activity.
- *(docs)* Tests wait for the UI to be idle before acting or asserting, so you do not write sleeps.
- *(general knowledge)* You find nodes by text, by tag (`Modifier.testTag`) or by other properties of the *semantics* tree, the same information that accessibility services use. That is a good reason to give icons a `contentDescription`: it helps users and tests.
- *(docs)* **The two-clocks warning.** Do not mix `createComposeRule` with `runTest` in one test: they have separate clocks. If you need coroutine control inside a UI test, the docs point to `runComposeUiTest` and the v2 testing APIs; read that page before you start.

(docs, from the Hilt chapter's sources) Tests that use Hilt itself need `hilt-android-testing`, test compilers on `kspAndroidTest` and a test activity marked `@AndroidEntryPoint`. Because this app uses constructor injection, none of the tests above needs it.

A Real Migration, and Its Test

Android Development with Compose: Architecture & Data 7 told you to uninstall the app after changing the entity. Real users cannot do that. Suppose you want to let notes be **pinned**: add a column, raise the version to 3, and write a migration.

```
@Entity(
    tableName = "notes",
    indices = [Index(value = ["remoteId"], unique = true)]
)
data class NoteEntity(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
```

```
    val text: String,
    val createdAt: Long = System.currentTimeMillis(),
    val remoteId: Long? = null,
    @ColumnInfo(defaultValue = "0") val pinned: Boolean = false      // new in version 3
)

@Database(entities = [NoteEntity::class], version = 3, exportSchema = true)
abstract class AppDatabase : RoomDatabase() { /* ... */ }
```

```
import androidx.room.migration.Migration
import androidx.sqlite.db.SupportSQLiteDatabase

val MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL("ALTER TABLE notes ADD COLUMN pinned INTEGER NOT NULL DEFAULT 0")
    }
}

// Where the database is built:
Room.databaseBuilder(context, AppDatabase::class.java, "notes.db")
    .addMigrations(MIGRATION_2_3)
    .build()
```

- *(docs)* Room exports the schema as JSON at compile time; keep those files in version control so older versions can be recreated for testing. The location is set through Room's Gradle plugin (check its current id and syntax in the docs).
- *(general knowledge)* `@ColumnInfo(defaultValue = "0")` should match the SQL default, so Room's schema validation agrees with what the migration produced.
- The code shows the Room 2.x form. *(docs, from Chapter 4)* In Room 3.0, migrations use `SQLiteConnection` instead of `SupportSQLiteDatabase` and the package is `androidx.room3`.

```
// app/src/androidTest/.../MigrationTest.kt
@RunWith(AndroidJUnit4::class)
class MigrationTest {

    private val dbName = "migration-test"

    @get:Rule
    val helper = MigrationTestHelper(
        InstrumentationRegistry.getInstrumentation(),
```

```
            AppDatabase::class.java
        )

        @Test
        fun migrate2To3_keepsNotesAndAddsPinned() {
            // 1. Create the database at the OLD version and insert a row with raw SQL
            helper.createDatabase(dbName, 2).apply {
                execSQL("INSERT INTO notes (id, text, createdAt) VALUES (1, 'Hi', 0)")
                close()
            }

            // 2. Run the migration; Room validates the resulting schema
            val db = helper.runMigrationsAndValidate(dbName, 3, true, MIGRATION_2_3)

            // 3. You check the DATA yourself: the helper only validates the schema
            db.query("SELECT text, pinned FROM notes WHERE id = 1").use { cursor ->
                assertTrue(cursor.moveToFirst())
                assertEquals("Hi", cursor.getString(0))
                assertEquals(0, cursor.getInt(1))
            }
        }
    }
```

(docs) The helper creates the old database, you insert a row with raw SQL, `runMigrationsAndValidate` migrates and validates the schema, and then **you** check the data: the helper does not check that your rows survived. The test above does exactly that.

What Makes a Test Worth Having

- **Test behaviour, not wiring.** "Adding an empty note shows a message" outlives refactoring; "the ViewModel calls `update` twice" does not *(general knowledge)*.
- **Test the edges.** Empty input, a failed network call, an old database version.
- **No sleeping.** Use the test dispatcher's virtual time and the Compose rule's idle waiting instead of real delays *(general knowledge, and the docs on idle waiting)*.
- **A test you never saw fail is unproven.** Break the code on purpose once and watch the test catch it *(general knowledge)*.

> **COMPARED WITH THE CLASSIC UI COURSE**

Android Development: Production & Publishing 2 teaches unit tests, instrumented tests, Espresso and Compose UI testing. Here Espresso (for View-based screens) gives way to Compose's own test APIs, and the new material is the ViewModel dispatcher rule, the two-clocks warning, and a worked migration test.

Try It Yourself

EXERCISE 1

Add the dependencies, the `MainDispatcherRule` and the two ViewModel tests, and run them. Then add a third test that checks the sort order setting changes the order of the notes. Report whether they pass and how long they took.

View solution

EXERCISE 2

Write the two Compose UI tests and a third that checks the Refresh button is disabled while `isRefreshing` is true. Run them on an emulator and report what you needed to add (the tag, the manifest dependency, anything else).

View solution

EXERCISE 3

Add the `pinned` column, raise the database to version 3, write the migration and the migration test, and set up schema export. Then break the migration on purpose (a wrong default, say) and confirm the test fails. Report how you configured the schema location.

View solution

NOT YET VERIFIED

Checked in the docs (two searches, summaries only): the Compose test setup and manifest dependency; `createComposeRule` against `createAndroidComposeRule`; the find, click, assert pattern; idle waiting; Compose tests being instrumented; the `StandardTestDispatcher` default; the two-clocks warning and `runComposeUiTest`; exported schemas in version control and the Gradle plugin's schema

location; the `MigrationTestHelper` flow and the need to check data yourself; auto-migrations needing `exportSchema = true`; the `room-testing` and `room3-testing` artifacts; the Hilt testing setup and `Dispatchers.setMain` (earlier chapters).

General knowledge, not checked: the local against instrumented split; `backgroundScope`; `performTextInput` and `testTag`; the semantics tree remark; the Room Gradle plugin's id; the migration and its SQL; the column default and schema validation remark; the advice on what makes a test worth having; every import and the code as written.

Not verified: the `room-testing` constructor and artifact for your Room line; every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.

Sources: Test your Compose layout (developer.android.com), Configure the test environment with v2 testing APIs (developer.android.com), Common patterns (developer.android.com), Migrate your Room database (developer.android.com), MigrationTestHelper API reference (developer.android.com), Hilt testing guide (developer.android.com).

COMING UP

Chapter 3 is Background Work: WorkManager, for the jobs that must finish even if the user leaves the app, such as a periodic sync of the notes.

Android Development with Compose: Production & Publishing

Course 6 · Chapter 3 · Work That Outlives the Screen

Android Development with Compose: Architecture & Data 3 said coroutines in `viewModelScope` are right for work tied to a screen, and that work which must finish *whatever the user does* belongs to WorkManager. This chapter is that promise: syncing the notes in the background, even if the app is closed or the phone restarts.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked in the docs (two searches, summaries only): Requests: `OneTimeWorkRequest` for one-off work and `PeriodicWorkRequest` for work on an interval; for Kotlin, extend `CoroutineWorker` rather than `Worker`, because it has a suspending `doWork()` and WorkManager cancels the coroutine when the work is stopped; constraints are set with `setConstraints` (network type, battery not low, charging, idle and so on); `setInitialDelay` on a one-time request, and for a periodic request only the first run is delayed; a periodic request can take a flex interval; requests accept backoff criteria. **Unique work:** `enqueueUniquePeriodicWork` allows one active periodic request with a given name; `REPLACE` cancels the pending one and runs the new one, `KEEP` runs the new one only if none is pending. **Hilt:** `@HiltWorker` with `@AssistedInject` and `@Assisted` context and `WorkerParameters`; the application class implements `Configuration.Provider` and supplies a `HiltWorkerFactory`. **Foreground services:** from Android 14 (API 34) every foreground service needs a declared type and the matching permission besides `FOREGROUND_SERVICE`, otherwise `MissingForegroundServiceTypeException` on `startForeground()`; long-running workers targeting Android 14 or higher must specify a type, declared on the service element; `setForeground()` triggers a prerequisite check for the type; from Android 16 long-running workers can exhaust an app's job quota; WorkManager is documented for tasks under about 10 minutes and supports longer work through a foreground service. **Not found in the searches (general knowledge, not checked):** the 15-minute minimum for periodic work, exact backoff behaviour, `TestListenableWorkerBuilder` as written, the manifest merge snippet, and the Result handling in the worker. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

Choosing the Right Tool

WORK	USE
Tied to a screen; fine to cancel when the user leaves	A coroutine in `viewModelScope`

WORK	USE
Must finish even if the app is closed or the phone restarts	WorkManager *(docs: persistent work)*
Long (beyond about 10 minutes *(docs)*), user-visible	A foreground service, or a WorkManager worker that runs as one

(general knowledge) WorkManager is for work that is *deferrable*: it will run when the conditions are right, not necessarily right now. If it must run at an exact time, it is not the tool.

```
dependencies {
    implementation("androidx.work:work-runtime-ktx:<work-version>")     // check the cur
    androidTestImplementation("androidx.work:work-testing:<work-version>")
}
```

A Worker for the Notes App

```
import android.content.Context
import androidx.work.CoroutineWorker
import androidx.work.WorkerParameters

class SyncNotesWorker(
    appContext: Context,
    params: WorkerParameters
) : CoroutineWorker(appContext, params) {

    override suspend fun doWork(): Result {
        // Manual DI (Architecture & Data 9): the worker reaches the container through t
        val repository = (applicationContext as NotesApplication).container.notesReposit

        return when (repository.refresh()) {
            is RefreshResult.Success -> Result.success()
            is RefreshResult.NoConnection -> Result.retry()     // try again later, with
            else -> Result.failure()
        }
    }
}
```

- *(docs)* `CoroutineWorker` has a suspending `doWork()`, so the repository's `suspend` functions can be called directly, and the coroutine is cancelled if the work is stopped.
- *(general knowledge)* The result tells WorkManager what to do next: `success` (done), `failure` (do not retry) or `retry` (run again later, following the backoff policy). A "no connection" result is a retry; a bad response is not.
- *(general knowledge)* A worker has no ViewModel and no screen. It gets its dependencies from the Application (manual DI, as here) or from a factory (Hilt, below).

Scheduling It

```
import androidx.work.*
import java.util.concurrent.TimeUnit

fun scheduleNotesSync(context: Context) {
    val constraints = Constraints.Builder()
        .setRequiredNetworkType(NetworkType.CONNECTED)
        .setRequiresBatteryNotLow(true)
        .build()

    val request = PeriodicWorkRequestBuilder<SyncNotesWorker>(1, TimeUnit.HOURS)
        .setConstraints(constraints)
        .setBackoffCriteria(BackoffPolicy.EXPONENTIAL, 30, TimeUnit.SECONDS)
        .build()

    // One named job: calling this again must not create a second one
    WorkManager.getInstance(context).enqueueUniquePeriodicWork(
        "notes-sync",
        ExistingPeriodicWorkPolicy.KEEP,
        request
    )
}

// In NotesApplication.onCreate():  scheduleNotesSync(this)
```

- **Constraints** *(docs)*: the work waits until the network is connected and the battery is not low. The system, not you, decides when.
- **Unique work** *(docs)*: `KEEP` means calling `scheduleNotesSync` on every app start does not stack up copies; `REPLACE` would cancel and restart. Without a unique name you could end

up with dozens of sync jobs.

- **Backoff**: *(docs)* requests accept backoff criteria; *(general knowledge, not checked)* exponential backoff roughly doubles the wait after each retry from the starting delay.
- **The interval**: *(general knowledge, not checked in this search)* periodic work has a minimum interval of 15 minutes, so a one-hour sync is realistic and a one-minute sync is not possible. The system may also run it later than the interval; it is a "roughly hourly", not "at 10:00".

One-time work for something the user asked for:

```
// Work the user asks for: export all notes now, but only on a charger
val export = OneTimeWorkRequestBuilder<ExportNotesWorker>()
    .setConstraints(Constraints.Builder().setRequiresCharging(true).build())
    .setInitialDelay(10, TimeUnit.SECONDS)
    .build()
WorkManager.getInstance(context).enqueue(export)
```

(docs) `setInitialDelay` delays a one-time request; on a periodic request only the first run is delayed. `ExportNotesWorker` would be a second `CoroutineWorker`, calling Chapter 11's `exportNotesAsJson` from the previous course.

With Hilt

```
// If the app uses Hilt (Architecture & Data 10): the docs' pattern
@HiltWorker
class SyncNotesWorker @AssistedInject constructor(
    @Assisted appContext: Context,
    @Assisted params: WorkerParameters,
    private val repository: NotesRepository
) : CoroutineWorker(appContext, params) { /* doWork as above */ }

@HiltAndroidApp
class NotesApplication : Application(), Configuration.Provider {
    @Inject lateinit var workerFactory: HiltWorkerFactory

    override val workManagerConfiguration: Configuration
        get() = Configuration.Builder().setWorkerFactory(workerFactory).build()
}
```

(docs) The worker class is annotated `@HiltWorker`, its constructor uses `@AssistedInject`, and the application supplies a `HiltWorkerFactory` through `Configuration.Provider`. *(general knowledge)* WorkManager normally creates workers itself, by reflection, with a fixed constructor; the factory is the hook that lets Hilt supply the extra arguments. Check the current docs for whether the default WorkManager initializer must be disabled in the manifest in your version; that detail was not in the search results.

Testing a Worker

```
// app/src/androidTest/.../SyncNotesWorkerTest.kt
@RunWith(AndroidJUnit4::class)
class SyncNotesWorkerTest {
    @Test
    fun syncSucceeds() = runTest {
        val context = ApplicationProvider.getApplicationContext<Context>()
        val worker = TestListenableWorkerBuilder<SyncNotesWorker>(context).build()

        val result = worker.doWork()

        assertEquals(ListenableWorker.Result.success(), result)
    }
}
```

(general knowledge, not checked in these searches) `work-testing` provides builders that create a worker directly so you can call `doWork()` and check the result, and a way to run the whole scheduling machinery in a test. *(docs, in the first search's results)* The docs also have a page on integration tests with WorkManager; read it before testing constraints and periodic timing. This is an instrumented test, like the ones in Chapter 2.

Long Work and Foreground Services (described, not built)

A sync of a few notes is short. If a worker needs to run for a long time and the user should know about it (uploading a large file, say), the docs describe two routes. The chapter describes them; the next chapter builds the notification they need.

```
// Inside a CoroutineWorker: tell WorkManager to run as a foreground service
override suspend fun doWork(): Result {
    setForeground(createForegroundInfo())       // needs a notification: Chapter 4
    // ...the long job...
    return Result.success()
}
```

```
<!-- Manifest: a long-running worker needs a foreground service type (Android 14+) -->
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
<uses-permission android:name="android.permission.FOREGROUND_SERVICE_DATA_SYNC" />

<service
    android:name="androidx.work.impl.foreground.SystemForegroundService"
    android:foregroundServiceType="dataSync"
    tools:node="merge" />
```

- *(docs)* A long-running worker is allowed to run as a foreground service through `setForeground()`. From Android 14 you must declare a type (and its permission) or the system throws `MissingForegroundServiceTypeException`.
- *(docs)* From Android 14 the system checks the type's prerequisites when you call `setForeground()`; *(docs)* from Android 16 long-running workers can use up the app's job quota, so launching a foreground service directly is an option.
- *(general knowledge, not checked)* The right type depends on what the work does (data sync, media, location and others); the "Foreground service types" page lists them and the permission each needs. The manifest snippet above is the shape, with the `dataSync` type as an example; confirm it against the current page and your target API.
- A foreground service shows a notification. That is Chapter 4's subject, with the permission needed on newer Android versions.

Seeing What WorkManager Did

(general knowledge, not checked) The App Inspection tool in Android Studio has a Background Task Inspector that lists workers and their states, and a worker's state moves through `ENQUEUED`, `RUNNING` and `SUCCEEDED` (or `FAILED`, `CANCELLED`, `BLOCKED`). Because constraints decide

when, "why has it not run?" is usually "a constraint is not met"; check the inspector before blaming the code.

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Production & Publishing 3 teaches background work: WorkManager, services and alarms. The WorkManager core is the same; this edition uses `CoroutineWorker` with the repository from Architecture & Data, adds the unique-work and Hilt patterns, and describes the Android 14 foreground-service type rule. Alarms and classic services are not covered here.

Try It Yourself

EXERCISE 1

Add `SyncNotesWorker` and `scheduleNotesSync` to the notes app. Run the app, then use the Background Task Inspector to find the job. Report its state and which constraints it was waiting on. Then change the interval to the shortest you are allowed and report what happens.

View solution

EXERCISE 2

Write `ExportNotesWorker` as one-time work that only runs while charging. Enqueue it from a button, then plug in or unplug the emulator's charger (Extended controls) and report what you see. Then explain why `REPLACE` against `KEEP` matters for the periodic sync.

View solution

EXERCISE 3

Write the worker test, then add a second test where the fake repository returns `NoConnection` and check the result is a retry. Say what you had to change to give the worker a fake (the worker creates its own repository from the Application).

View solution

NOT YET VERIFIED

Checked in the docs (two searches, summaries only): the two request types; `CoroutineWorker` and its cancellation; `setConstraints` and the constraint kinds; `setInitialDelay` and its periodic behaviour; flex intervals; backoff criteria existing; `enqueueUniquePeriodicWork`, `KEEP` and `REPLACE`; the Hilt worker pattern; the Android 14 foreground-service type rule, the exception, the prerequisite check, the Android 16 job-quota note, and the roughly-10-minutes guidance.

General knowledge, not checked: the 15-minute periodic minimum; how exponential backoff grows; the `Result` mapping in the worker; the manual-DI worker; the Hilt factory explanation and the manifest initializer detail; `work-testing` and `TestListenableWorkerBuilder` as written; the manifest service snippet and the `dataSync` choice; the Background Task Inspector and the work states; every import and the code as written.

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

Sources: Define work requests (developer.android.com), Threading in CoroutineWorker (developer.android.com), WorkManager API reference (developer.android.com), Support for long-running workers (developer.android.com), Foreground service types are required (developer.android.com), Foreground service types (developer.android.com), Integration tests with WorkManager (developer.android.com).

COMING UP

Chapter 4 is Notifications & Permissions: channels, the runtime permission flow in Compose, handling a "no", and a reminder notification for a note.

Android Development with Compose: Production & Publishing

Course 6 · Chapter 4 · Asking Before You Interrupt

A notification interrupts someone, so Android makes you ask first. This chapter adds a reminder to a note: it creates a notification channel, asks for the notification permission in Compose, copes with a "no", and uses Android Development with Compose: Production & Publishing 3's WorkManager to fire the reminder later.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked in the docs (two searches, summaries only): Permission: Android 13 (API 33) added `POST_NOTIFICATIONS` as a runtime permission, applying to all apps running on Android 13 whatever their target; if denied, the app cannot send notifications unless it has an exemption; newly installed apps start with notifications off, while upgraded apps are typically pre-granted; before sending, call `areNotificationsEnabled()`; `notify` is annotated as requiring the permission. **Channels:** required on Android 8.0+; create them when the app starts, since creating an existing one does nothing; create before posting. **Compose request:** `rememberLauncherForActivityResult` with `ActivityResultContracts.RequestPermission()`, launching the permission behind a check for Android 13; no rationale screen is needed unless `shouldShowRequestPermissionRationale()` returns true; in Compose the current Activity can be taken from `LocalContext.current`. **Exact alarms:** `SCHEDULE_EXACT_ALARM` is a special access permission, user-revocable, for user-facing features; `USE_EXACT_ALARM` is for apps whose core function is alarms, timers or calendars; since Android 14 the first is denied by default for most new apps targeting Android 13+; call `canScheduleExactAlarms()` and otherwise send the user to `ACTION_REQUEST_SCHEDULE_EXACT_ALARM`. **Not found (general knowledge, not checked):** the PendingIntent and deep-link code, the notification builder as written, the settings intent, the small icon resource, Accompanist's permissions library (not searched), and the interplay with Chapter 3's foreground-service notifications. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

Channels

```
<!-- AndroidManifest.xml: declare the permission (Android 13+ asks the user at run time)
<uses-permission android:name="android.permission.POST_NOTIFICATIONS" />
```

```
import android.app.NotificationChannel
import android.app.NotificationManager
import android.content.Context

const val REMINDER_CHANNEL_ID = "note_reminders"

fun createNotificationChannels(context: Context) {
    // Channels exist on Android 8.0+ (API 26); minSdk 26 or higher needs no version che
    val channel = NotificationChannel(
        REMINDER_CHANNEL_ID,
        "Note reminders",
        NotificationManager.IMPORTANCE_DEFAULT
    ).apply { description = "Reminders you set on a note" }

    context.getSystemService(NotificationManager::class.java).createNotificationChannel(
}

// In NotesApplication.onCreate():  createNotificationChannels(this)
// (docs: create the channel at app start; creating an existing channel does nothing)
```

- *(docs)* Channels are required from Android 8.0. Create them at app start; making one that already exists does nothing.
- *(general knowledge)* The channel carries the user-visible name and the importance. After it is created, **you cannot change the user's choices**: they control sound and importance in system settings. Choose the importance carefully, and use one channel per kind of notification so users can silence reminders without silencing everything.

Posting a Notification

```
import android.Manifest
import android.app.PendingIntent
import android.content.Intent
import android.content.pm.PackageManager
import androidx.core.app.NotificationCompat
import androidx.core.app.NotificationManagerCompat
import androidx.core.content.ContextCompat

fun showReminder(context: Context, noteId: Long, text: String) {
    val openApp = PendingIntent.getActivity(
```

```
            context,
            noteId.toInt(),
            Intent(context, MainActivity::class.java).putExtra("noteId", noteId),
            PendingIntent.FLAG_IMMUTABLE or PendingIntent.FLAG_UPDATE_CURRENT
        )

        val notification = NotificationCompat.Builder(context, REMINDER_CHANNEL_ID)
            .setSmallIcon(R.drawable.ic_stat_note)
            .setContentTitle("Note reminder")
            .setContentText(text)
            .setContentIntent(openApp)
            .setAutoCancel(true)
            .build()

        // notify() is annotated as needing POST_NOTIFICATIONS (docs), so check first
        val allowed = Build.VERSION.SDK_INT < Build.VERSION_CODES.TIRAMISU ||
            ContextCompat.checkSelfPermission(context, Manifest.permission.POST_NOTIFICATIONS
                PackageManager.PERMISSION_GRANTED
        if (allowed) {
            NotificationManagerCompat.from(context).notify(noteId.toInt(), notification)
        }
    }
```

- *(docs)* `notify` needs `POST_NOTIFICATIONS` on Android 13+, so the code checks first. *(docs)* `areNotificationsEnabled()` is the other check the guide mentions.
- *(general knowledge)* The `PendingIntent` says what happens on a tap. `FLAG_IMMUTABLE` is the safe default on current Android; the extra `noteId` lets `MainActivity` open that note. A different request code per note stops reminders from overwriting each other's intents.
- *(general knowledge)* A notification needs a small icon (`ic_stat_note` here is a drawable you supply) or it will not appear properly.

Asking for the Permission in Compose

```
import android.Manifest
import android.os.Build
import androidx.activity.compose.rememberLauncherForActivityResult
import androidx.activity.result.contract.ActivityResultContracts
import androidx.compose.runtime.*

@Composable
```

```
fun NotificationPermissionButton(onResult: (Boolean) -> Unit) {
    val launcher = rememberLauncherForActivityResult(
        ActivityResultContracts.RequestPermission(),
        onResult                                    // true if the user allowed it
    )

    Button(onClick = {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
            launcher.launch(Manifest.permission.POST_NOTIFICATIONS)
        } else {
            onResult(true)                          // before Android 13 there is nothin
        }
    }) { Text("Allow reminders") }
}
```

- *(docs)* The sample uses exactly this launcher with `RequestPermission()`, behind an Android 13 version check.
- **When to ask.** *(general knowledge)* At the moment the user taps "Remind me", not at app start, because then they know why. A system dialog without context is easy to refuse.
- **Rationale.** *(docs)* You do not need to show a rationale screen unless `shouldShowRequestPermissionRationale()` returns true; the method needs the Activity, available through `LocalContext.current`.

When the User Says No

```
import android.content.Intent
import android.net.Uri
import android.provider.Settings

// Ask at the moment the user wants a reminder, not at app start.
// If they said no, explain and offer the system settings page instead of asking again.
fun openNotificationSettings(context: Context) {
    val intent = Intent(Settings.ACTION_APP_NOTIFICATION_SETTINGS)
        .putExtra(Settings.EXTRA_APP_PACKAGE, context.packageName)
    context.startActivity(intent)
}
```

(docs) A denial means no notifications. *(general knowledge, not checked)* After repeated denials the system may stop showing the dialog, so asking again does nothing. The respectful design is: the app keeps working (the reminder is simply not set), explain once why it would help, and offer a button to the system settings page. Never nag, and never make a core feature depend on the permission.

The Reminder, Built on WorkManager

```
class ReminderWorker(
    appContext: Context,
    params: WorkerParameters
) : CoroutineWorker(appContext, params) {
    override suspend fun doWork(): Result {
        val id = inputData.getLong("noteId", -1)
        val text = inputData.getString("text") ?: return Result.failure()
        showReminder(applicationContext, id, text)
        return Result.success()
    }
}

// "Remind me in an hour": inexact, so no exact-alarm permission is needed
fun scheduleReminder(context: Context, noteId: Long, text: String, delayMinutes: Long) {
    val request = OneTimeWorkRequestBuilder<ReminderWorker>()
        .setInitialDelay(delayMinutes, TimeUnit.MINUTES)
        .setInputData(workDataOf("noteId" to noteId, "text" to text))
        .build()
    WorkManager.getInstance(context).enqueueUniqueWork(
        "reminder-$noteId", ExistingWorkPolicy.REPLACE, request
    )
}
```

- *(docs, Chapter 3)* `setInitialDelay` delays a one-time request; unique work with `REPLACE` means setting a new reminder for the same note replaces the old one.
- *(general knowledge)* This is **inexact**: the system may run it a little late to save battery, which is fine for "remind me in an hour".

Exact Alarms (described, not used)

```
// Only if a feature truly needs an exact time (the docs say: user-facing features)
val alarmManager = context.getSystemService(AlarmManager::class.java)
if (alarmManager.canScheduleExactAlarms()) {
    // schedule the exact alarm
} else {
    context.startActivity(Intent(Settings.ACTION_REQUEST_SCHEDULE_EXACT_ALARM))
}
```

- *(docs)* Exact timing needs `SCHEDULE_EXACT_ALARM` (user-revocable, for user-facing features) or `USE_EXACT_ALARM` (only for apps whose core function is alarms, timers or calendars).
- *(docs)* Since Android 14 the first is denied by default for most new apps targeting Android 13+, so you must check `canScheduleExactAlarms()` and, if false, send the user to the settings screen.
- *(general knowledge)* A notes app's "remind me later" does not need to the second, so it should not request exact alarms: asking for a permission you do not need costs trust, and Google Play reviews such requests *(checked for real in Chapter 7)*.

Permissions in General

KIND	HOW THE USER GRANTS IT	EXAMPLE HERE
Normal (install time)	Granted automatically	`INTERNET` *(general knowledge)*
Runtime	A system dialog, at the moment you ask	`POST_NOTIFICATIONS` *(docs)*
Special access	A settings screen you send them to	`SCHEDULE_EXACT_ALARM` *(docs)*

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Production & Publishing 4 teaches notifications and permissions. The notification code is the same (it does not depend on Compose); the Compose edition requests permission with `rememberLauncherForActivityResult` instead of the Activity's callbacks, and builds the reminder on WorkManager.

Try It Yourself

EXERCISE 1

Add the manifest permission, the channel and `showReminder`, with a debug button that posts one immediately. Test on Android 13+ and on an older emulator if you have one. Report what happens before and after granting, and where the channel appears in system settings.

View solution

EXERCISE 2

Build the permission flow: tapping "Remind me" asks once, and a denial shows a message with a button to the settings page. Deny twice and report what the system does the third time.

View solution

EXERCISE 3

Add `ReminderWorker` and `scheduleReminder`, set a reminder for one minute ahead, and tap the notification. Make `MainActivity` read `noteId` and open that note. Say why you would not use an exact alarm here.

View solution

NOT YET VERIFIED

Checked in the docs (two searches, summaries only): the Android 13 runtime permission and its all-apps scope; denial blocking notifications; new against upgraded installs; `areNotificationsEnabled()`; the `notify` annotation; channels on Android 8.0+ and creating them at start; the Compose launcher sample and version check; rationale and `LocalContext.current`; the exact-alarm permissions, the Android 14 default and `canScheduleExactAlarms()`.

General knowledge, not checked: the notification builder, `PendingIntent` flags and the deep-link extra; the small icon requirement; the channel choices being the user's afterwards; the settings intent; the system stopping the dialog after repeated denials; the worker and its inexact timing; the permission-kinds table's install-time row; the Play review remark; every import and the code as written.

Not verified: Accompanist's permissions library (not searched); foreground-service notification rules (the second search did not return them); every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.

Sources: Notification runtime permission (developer.android.com), Create a notification, Compose (developer.android.com), Display time-sensitive notifications (developer.android.com), Behavior

changes: all apps, Android 13 (developer.android.com), Schedule exact alarms are denied by default (developer.android.com), Schedule alarms (developer.android.com).

COMING UP

Chapter 5 is App Security: shrinking and obfuscating with R8, secure storage, biometric sign-in, network security, and keeping secrets out of the code.

Android Development with Compose: Production & Publishing

Course 6 · Chapter 5 · Locking the Door

Security advice goes out of date faster than most Android topics, and this chapter is one of the three the course outline flagged as risky. So it is open about what it checked and how old things might be: read the evidence box first, and treat the specific classes and settings as **things to confirm against the current docs** before you ship.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked in the docs on 2026-11-22 (four searches, summaries only, results of mixed age): R8: `isMinifyEnabled` for code and `isShrinkResources` for resources in the release build type (legacy DSL); resource shrinking only works with code shrinking; use `proguard-android-optimize.txt`, because the older `proguard-android.txt` does not enable R8's optimisations; your own rules go in a file such as `proguard-rules.pro`; keep rules from all sources are merged; **newer AGP (9.3 and higher, per the docs) has an** `optimization { enable = true }` **block and** `.keep` **files in** `src/<variant>/keepRules`. **Secrets:** the security checklist advises against keys in source code and mentions the secrets-gradle-plugin; `local.properties` is described as reserved for AGP, so use a separate properties file for your own values. **Storage:** `EncryptedSharedPreferences` **is deprecated** (security-crypto 1.1.0), with the Jetpack notes pointing to platform APIs and direct use of the Android Keystore; its files should be excluded from Auto Backup because restoring fails without the key; the checklist advises the Keystore for keys and a tool such as Tink for stored keys; Keystore key material never enters the app process and Keystore work should stay off the main thread; DataStore has a `datastore-tink` artifact with an AEAD serializer; sensitive data belongs in internal storage. **Biometrics:** `BiometricPrompt` with an executor, a callback and a `PromptInfo` using `setAllowedAuthenticators(BIOMETRIC_STRONG or DEVICE_CREDENTIAL)`; a `CryptoObject` ties a key to the prompt, and then fallback to device credentials is not allowed for that key; the biometric codelab found is itself marked deprecated; no Compose wrapper was in the results. **Network:** cleartext is off by default from API 28; per-domain opt-ins in a `domain-config`; `debug-overrides` do not affect release; a `pin-set` holds pins with an optional expiration date. **Backup:** `allowBackup`, and `dataExtractionRules` on Android 12+ with separate cloud and device-transfer rules. **Not found:** `mapping.txt` and retracing stack traces; the secrets plugin's setup; a full pin example. **General knowledge, not checked:** everything labelled so below. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

Shrinking and Obfuscating with R8

```
// app/build.gradle.kts: shrink, obfuscate and optimise the RELEASE build
android {
    buildTypes {
        release {
            isMinifyEnabled = true          // R8: shrink, obfuscate, optimise code
            isShrinkResources = true        // needs isMinifyEnabled = true (docs)
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),    // not the ol
                "proguard-rules.pro"                                        // your own k
            )
        }
    }
}
// Newer AGP (docs: 9.3 and higher) has an optimization { enable = true } block and keep
// src/<variant>/keepRules as .keep files. Check which one your AGP uses before copying
```

```
# proguard-rules.pro: add rules only when the release build breaks, and keep them narrow
# Example shape (general knowledge): keep a class that is created by reflection
-keep class com.example.notes.data.RemoteNoteDto { *; }
```

- *(docs)* R8 removes unused code and resources, shortens names and optimises, in the release build. Resource shrinking depends on code shrinking.
- *(docs)* The default file you pass matters: the `-optimize` one turns the optimisations on.
- *(general knowledge)* Code that is found by name at run time (reflection) can be removed or renamed by R8 and then crash only in release builds. That is what keep rules are for. Many libraries ship their own rules, so add your own only when something breaks, and keep them narrow.
- *(general knowledge, not found in the searches)* Obfuscated stack traces need the `mapping.txt` file R8 writes with each release build; keep it for every version you publish (Play can also use it). The searches did not cover retracing, so confirm the details in the docs.
- **Always test the release build** *(general knowledge)*: a debug build does not run R8, so a keep-rule bug shows up only there.

Keeping Keys Out of the Code

```
// app/build.gradle.kts: read a key from a file that is NOT in version control (general
// Docs: do not put keys in source; use something like the Secrets Gradle Plugin.
// Docs warn: local.properties is reserved for AGP; for your own values, make a separate
// properties file and load it yourself (or follow the plugin's own docs).
val secrets = java.util.Properties().apply {
    rootProject.file("secrets.properties").takeIf { it.exists() }?.inputStream()?.use(::
}
android {
    buildFeatures { buildConfig = true }
    defaultConfig {
        buildConfigField("String", "NOTES_API_KEY", "\"${secrets.getProperty("NOTES_API_
    }
}
// .gitignore: secrets.properties
// The key is still in the APK: anything shipped in the app can be extracted. This only
// out of your repository. A key that must stay secret belongs on a server, not in the a
```

(docs) Do not put keys in source code. *(general knowledge)* Keeping a key out of the repository is not the same as keeping it secret: whatever is inside the APK can be read by someone determined. If a key must really stay secret, it lives on a server that your app asks, and the server decides what the app is allowed to do.

Secure Storage

- **Start by asking whether you need to store it.** *(general knowledge)* Not storing a secret is the strongest protection.
- **Internal storage only.** *(docs)* Sensitive data belongs in the app's sandboxed internal storage, whatever Android version you target.
- `EncryptedSharedPreferences` **is deprecated** *(docs)*. Older tutorials, and Android Development: Production & Publishing 5's era of advice, use it. The Jetpack notes now point to platform APIs and direct use of the Android Keystore. Check the current docs for the replacement for your case before relying on any wrapper library.
- **The Keystore** *(docs)*: keys are generated and used inside it and never enter your process; use it off the main thread. A common design (docs: keys with a tool such as Tink) is a Keystore-held key that encrypts the data you store.

- **DataStore with Tink** *(docs)*: a `datastore-tink` artifact wraps the serializer with AEAD encryption, with a unique associated-data value per file. *(general knowledge)* That suits the Proto DataStore of Architecture & Data 6 when a setting is sensitive. Your notes themselves are in a Room database that is **not encrypted** by default; encrypting it is a bigger decision (it needs a third-party library or similar, not checked here), so decide whether your notes really need it.

Biometric Unlock

```
import androidx.biometric.BiometricManager
import androidx.biometric.BiometricManager.Authenticators.BIOMETRIC_STRONG
import androidx.biometric.BiometricManager.Authenticators.DEVICE_CREDENTIAL
import androidx.biometric.BiometricPrompt
import androidx.core.content.ContextCompat
import androidx.fragment.app.FragmentActivity

// Docs show the View-style API: an executor, a callback, a PromptInfo. There is no Compo
// in the sources checked, so a small function bridges it. MainActivity must be a Fragme
// (ComponentActivity alone is not enough for BiometricPrompt: general knowledge, check)
fun showBiometricPrompt(activity: FragmentActivity, onSuccess: () -> Unit, onFail: () ->
    val authenticators = BIOMETRIC_STRONG or DEVICE_CREDENTIAL
    val canAuthenticate = BiometricManager.from(activity).canAuthenticate(authenticators
    if (canAuthenticate != BiometricManager.BIOMETRIC_SUCCESS) { onFail(); return }

    val prompt = BiometricPrompt(
        activity,
        ContextCompat.getMainExecutor(activity),
        object : BiometricPrompt.AuthenticationCallback() {
            override fun onAuthenticationSucceeded(result: BiometricPrompt.Authenticatio
            override fun onAuthenticationError(errorCode: Int, errString: CharSequence) =
        }
    )

    val info = BiometricPrompt.PromptInfo.Builder()
        .setTitle("Unlock your notes")
        .setSubtitle("Use your fingerprint, face or screen lock")
        .setAllowedAuthenticators(authenticators)       // as in the docs' example
        .build()
```

```
    prompt.authenticate(info)       // no CryptoObject: this is a simple "who is holding
}
```

```
// A note-level lock: the prompt gates the screen. It is a convenience barrier, not encry
// Docs: to tie a Keystore key to a biometric you pass a CryptoObject, and then you cann
// allow fallback to device credentials; that is the stronger, more complicated route.
// Docs (Keystore): key material never enters your app process; avoid Keystore work on t
```

- *(docs)* The prompt is built with an executor, a callback and a `PromptInfo`; `BIOMETRIC_STRONG or DEVICE_CREDENTIAL` allows a fingerprint, face or the screen lock.
- *(docs)* The biometric codelab I found is marked deprecated, so treat the sample as a pattern, and confirm the current API in the "Show a biometric authentication dialog" page.
- *(general knowledge)* BiometricPrompt works with a `FragmentActivity`, so check what your `MainActivity` extends. The check with `canAuthenticate` avoids showing a prompt the device cannot answer.
- *(docs)* A `CryptoObject` binds a Keystore key to the prompt, which is cryptographically stronger than a screen lock, but a key like that cannot allow device-credential fallback. Choose by what you protect: a "private notes" screen can use the simple gate; unlocking encrypted data wants the key.

Network Security

```
<!-- res/xml/network_security_config.xml (docs: cleartext is already off by default from
<network-security-config>
    <base-config cleartextTrafficPermitted="false" />

    <!-- Debug builds only: allow your local test server (docs: debug-overrides does not
    <debug-overrides>
        <trust-anchors>
            <certificates src="user" />
        </trust-anchors>
    </debug-overrides>
</network-security-config>

<!-- Manifest: <application android:networkSecurityConfig="@xml/network_security_config"
```

```
<!-- Pinning, described (docs: a pin-set holds pins and an optional expiration date):
     <domain-config>
         <domain includeSubdomains="true">api.example.com</domain>
         <pin-set expiration="2027-12-31">
             <pin digest="SHA-256">...base64 of the public key hash...</pin>
         </pin-set>
     </domain-config>
     A wrong or stale pin locks the app out of its own server, so it is risky. -->
```

- *(docs)* Cleartext (non-HTTPS) traffic is off by default from API 28; opt in only per domain, and only if you must. The sample repo from Architecture & Data 8 talks to an HTTPS API, so none is needed.
- *(docs)* `debug-overrides` applies in debug builds without changing release.
- **Pinning** *(docs: a pin-set with an optional expiration)* makes the app trust only certain keys. *(general knowledge)* It protects against a rogue certificate authority but breaks the app if the server's key changes and your pins are stale, so use it only if you control the server and can update the app in time. Described here, not recommended for a notes app.

```
<!-- Manifest: choose what is backed up (docs: dataExtractionRules on Android 12+) -->
<application
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules" ... />

<!-- res/xml/data_extraction_rules.xml: exclude sensitive files (docs: cloud and device
<data-extraction-rules>
    <cloud-backup>
        <exclude domain="sharedpref" path="secure_prefs.xml" />
        <exclude domain="file" path="datastore/secure_settings.preferences_pb" />
    </cloud-backup>
    <device-transfer>
        <exclude domain="sharedpref" path="secure_prefs.xml" />
        <exclude domain="file" path="datastore/secure_settings.preferences_pb" />
    </device-transfer>
</data-extraction-rules>
```

- *(docs)* `allowBackup` controls participation; on Android 12+ `dataExtractionRules` separates cloud backup from device transfer, and you can exclude sensitive files.
- *(docs)* Encrypted preferences and Keystore-bound ciphertext should be excluded: after a restore the key is missing and the data cannot be read.
- *(general knowledge)* Also check that the files you name really exist under those paths; a rule that excludes nothing gives a false sense of safety.

A Short Security Checklist for the Notes App

RISK	WHAT THIS CHAPTER DOES
Reverse engineering	R8 on release (raises effort, not a guarantee *(general knowledge)*)
Keys in the repository	Separate untracked properties file; server-side for real secrets
Someone picks up the phone	Biometric gate on private notes
Data on the device	Internal storage; encrypt only what needs it; exclude it from backups
Traffic	HTTPS only (the default)

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Production & Publishing 5 teaches app security: R8, secure storage, biometric authentication, network security and keeping keys out of code. The topics match; this edition says plainly that `EncryptedSharedPreferences` is now deprecated, uses the `-optimize` default file, and notes the newer AGP keep-rules layout.

Try It Yourself

EXERCISE 1

Turn on R8 for the release build, build a release APK or bundle, and run it. Report whether it works. If something breaks, find it in the logs and write the narrowest keep rule that fixes it. Say which AGP version and which DSL form you used.

View solution

EXERCISE 2

Add the biometric gate to a "private notes" screen. Test on an emulator with a fingerprint enrolled and with none. Report what `canAuthenticate` returns in each case and what your app does.

View solution

EXERCISE 3

Audit your own app against the checklist table: list every place it stores data, every network call, whether backups include it, and any key in the code. Then check the current docs for the replacement for `EncryptedSharedPreferences` and write what you found.

View solution

NOT YET VERIFIED

Checked in the docs on 2026-11-22 (four searches, summaries only): the R8 settings and default file, keep rule merging and the newer AGP form; the advice against keys in source and `local.properties` being reserved; the `EncryptedSharedPreferences` deprecation and backup exclusion; Keystore key handling and the Tink advice; `datastore-tink`; internal storage; `BiometricPrompt` and `CryptoObject` behaviour; the deprecated codelab; cleartext defaults, `debug-overrides` and `pin-set`; `allowBackup` and `dataExtractionRules`.

General knowledge, not checked: why release-only crashes happen and what reflection has to do with it; `mapping.txt` and retracing; the properties-file code and the claim that anything in the APK can be extracted; the biometric code as written and the `FragmentActivity` requirement; the pinning trade-offs; the Room database being unencrypted by default; the backup file paths; the checklist table; every import and the code as written.

Not verified: the secrets plugin's setup; a full pin example; the replacement for `EncryptedSharedPreferences` (the docs point to platform APIs and the Keystore, but no replacement code was seen); the current AGP DSL for your version; every listing and solution (unrun); the exercises (not run by me). **Re-check the live docs before you ship: security advice moves.**

Sources: Enable app optimization with R8 (developer.android.com), About keep rules (developer.android.com), Security checklist (developer.android.com), EncryptedSharedPreferences API reference (developer.android.com), Android Keystore system (developer.android.com), Show a biometric authentication dialog (developer.android.com), Network security configuration (developer.android.com), Security recommendations for backups (developer.android.com).

COMING UP

Chapter 6 is Performance & Optimisation: profiling with Android Studio, memory leaks, ANRs and baseline profiles, with R8 from this chapter as one of the biggest wins.

Android Development with Compose: Production & Publishing

Course 6 · Chapter 6 · Fast Enough, Proved

"Make it faster" without a measurement is guessing. Android Development with Compose: Production & Publishing 1 already quoted the docs: do not judge performance by recomposition counts alone. This chapter is the other half: the tools that tell you what is actually slow, the two failures users notice most (freezes and crashes from running out of memory), and baseline profiles, the biggest free startup win.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked in the docs (three searches, summaries only): Baseline profiles and Macrobenchmark: for Compose apps Google recommends a Macrobenchmark-based baseline profile covering critical user journeys, and says to test that a custom profile helps; the current setup is the Baseline Profile module template, or the Macrobenchmark template with the Baseline Profile Gradle plugin; profiles can cover scrolling, animation and navigation, not only startup; Compose screens signal when fully drawn with `ReportDrawn`, `ReportDrawnWhen` and `ReportDrawnAfter`; jank is measured with `FrameTimingMetric` on a `LazyColumn`, with `testTagAsResourceId` enabled so UI Automator can find Compose elements; the docs list AGP 8.0.0, Macrobenchmark 1.5.0 and ProfileInstaller 1.4.1 as requirements (*versions as the docs showed them: check the current ones*); the target must be profileable and include ProfileInstaller 1.3 or later; Macrobenchmark is recommended for measuring because it repeats runs; startup is compared with `StartupTimingMetric` and `CompilationMode` (one docs example: 229.0 ms with a baseline profile against 324.8 ms without compilation); generating on a Gradle Managed Device needs an `aosp` image, though newer releases allow Android 13+ without root; Compose composition tracing can be enabled with `androidx.benchmark.fullTracing.enable`, and tracing during cold start distorts timing. **Android vitals:** the vitals dashboard shows stack traces for field ANRs with suggested fixes; the core vitals include crash rate, ANR rate, excessive battery use and excessive partial wake locks. **ANRs and StrictMode:** an ANR is triggered when the UI thread is blocked for too long; common causes are slow I/O, long calculations and synchronous binder calls on the main thread; keep disk and network work off the main thread; StrictMode has a `ThreadPolicy` (one thread) and a `VmPolicy` (the whole process), built with `detectAll()`; the `VmPolicy` has `detectActivityLeaks()` and related leak detectors; enable StrictMode only on debug builds, with a crashing penalty if you want violations hard to miss. **Memory:** capture a heap dump from the Profiler tab; recent Android Studio releases include a LeakCanary profiler task; Android Studio can open only heap dumps captured by its own Memory Profiler. **Not found:** the exact ANR timeout (the results did not state the figure, so none is given here), and the Profiler's CPU and frame views. **General knowledge, not checked:** everything labelled so below. **All code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen on your install.

The Rule: Measure, Change One Thing, Measure Again

SYMPTOM	FIRST TOOL
Slow to start	Macrobenchmark `StartupTimingMetric` *(docs)*, then a baseline profile
Janky scrolling	Macrobenchmark `FrameTimingMetric` *(docs)*; trace file in Android Studio
Freezes ("not responding")	StrictMode in debug *(docs)*; vitals ANR traces in production *(docs)*
Memory grows, then crashes	Profiler heap dump *(docs)*, leak detectors
Recomposing too much	Chapter 1's checks; composition tracing *(docs)*

(general knowledge) Measure on a **release-like build on a real device** where you can: a debug build, an emulator on a fast laptop and the R8-optimised release build (Android Development with Compose: Production & Publishing 5) can differ a lot. The docs say the target must be profileable for Macrobenchmark.

Freezes and ANRs

- *(docs)* An ANR happens when the UI thread is blocked for too long. Slow I/O, long calculations and synchronous binder calls on the main thread are the usual causes. (The exact timeout was not in the search results; check the ANR page for the current figures.)
- *(docs)* The fix is structural: keep disk and network work off the main thread. The notes app already does this by design: Room's suspend functions and Flow, the repository with `withContext`, Retrofit's suspend calls.

```
// Main-safe by construction (Architecture & Data 3): the repository switches threads its
class DefaultNotesRepository(
    private val dao: NoteDao,
    private val ioDispatcher: CoroutineDispatcher = Dispatchers.IO
) {
    suspend fun exportAll(): String = withContext(ioDispatcher) {
        // heavy file/JSON work: never on the main thread
        buildExportText(dao.getAll())
    }
}
```

```
// NotesApplication.onCreate(): catch accidental main-thread work and leaks in DEBUG bui
if (BuildConfig.DEBUG) {
    StrictMode.setThreadPolicy(
        StrictMode.ThreadPolicy.Builder()
            .detectAll()
            .penaltyLog()           // docs suggest a crashing penalty so violations are
            .build()
    )
    StrictMode.setVmPolicy(
        StrictMode.VmPolicy.Builder()
            .detectActivityLeaks()
            .detectLeakedClosableObjects()
            .detectLeakedRegistrationObjects()
            .detectLeakedSqlLiteObjects()
            .penaltyLog()
            .build()
    )
}
```

- *(docs)* `ThreadPolicy` watches one thread (the one you set it on), `VmPolicy` watches the process. `detectAll()` turns on every thread check.
- *(docs)* Enable StrictMode on debug builds only. *(general knowledge)* `penaltyLog()` writes a Logcat line; `penaltyDeath()` crashes the app, loud but effective on a small project.
- *(docs)* In production, the vitals dashboard shows ANR stack traces from real users.

Memory Leaks

(general knowledge) A leak is an object that is no longer needed but is still reachable, so it cannot be collected. In Android the typical case is a long-lived object (a singleton, a static field) holding an Activity.

```
// A classic leak: a long-lived object holds a short-lived one
object NotesCache {
    var screenCallback: (() -> Unit)? = null      // a global (long-lived) variable...
}

@Composable
fun NotesScreen() {
```

```
    val context = LocalContext.current            // an Activity
    NotesCache.screenCallback = { Toast.makeText(context, "Hi", Toast.LENGTH_SHORT).show
    // ...the lambda captures the Activity, and the object lives as long as the app does
    // the Activity cannot be garbage-collected after rotation or finish().
}

// The fix: do not park UI references in long-lived objects; clear them when the screen
// disposed, e.g. DisposableEffect(Unit) { onDispose { NotesCache.screenCallback = null
```

- *(docs)* `VmPolicy.detectActivityLeaks()` is the StrictMode check for exactly this.
- *(docs)* Take a **heap dump** from the Profiler: you can see which objects exist and what holds them. Android Studio opens only its own Memory Profiler dumps.
- *(docs)* Recent Android Studio versions include a LeakCanary profiler task that analyses leaks inside the IDE. *(general knowledge)* LeakCanary is also a library you can add to debug builds; the search did not cover its setup, so check its own docs.
- *(general knowledge)* A ViewModel does not leak the screen unless you hand it a Context or a view; that is the reason the ViewModel Chapter of the previous course told you not to.

Baseline Profiles

(docs) Android compiles app code as needed, so the first run is slower. A baseline profile lists the classes and methods your users hit most, so they are compiled ahead of time. Compose gains a lot because its library code is otherwise warmed up on first use.

```
// The generator (docs: BaselineProfileRule, in the Baseline Profile module)
@RunWith(AndroidJUnit4::class)
class BaselineProfileGenerator {
    @get:Rule val rule = BaselineProfileRule()

    @Test
    fun generate() = rule.collect(packageName = "com.example.notes") {
        pressHome()
        startActivityAndWait()
        // Walk the journeys users take most: scroll the list, open a note
    }
}
```

```
// Tell the benchmark (and the system) when the screen is really ready (docs: ReportDrawn
@Composable
fun NotesScreen(state: NotesUiState) {
    ReportDrawnWhen { !state.isLoading }
    // ...
}
```

```
// A Macrobenchmark module (docs: use the Baseline Profile / Macrobenchmark module templa
@RunWith(AndroidJUnit4::class)
class StartupBenchmark {
    @get:Rule val benchmarkRule = MacrobenchmarkRule()

    @Test
    fun startupNoCompilation() = startup(CompilationMode.None())

    @Test
    fun startupBaselineProfile() = startup(CompilationMode.Partial())     // docs' compar

    private fun startup(mode: CompilationMode) = benchmarkRule.measureRepeated(
        packageName = "com.example.notes",
        metrics = listOf(StartupTimingMetric()),
        iterations = 10,
        startupMode = StartupMode.COLD,
        compilationMode = mode
    ) {
        pressHome()
        startActivityAndWait()
    }
}
```

- *(docs)* Set up with the **Baseline Profile module template** in Android Studio (or the Macrobenchmark template and the Baseline Profile Gradle plugin). Generation is a test that walks critical journeys; the plugin turns the result into a profile in your app.
- *(docs)* The comparison test above uses `StartupTimingMetric` and `CompilationMode`. A docs example showed 229.0 ms with a baseline profile and 324.8 ms with no compilation: **an illustration of the method, not a promise for your app.** Measure yours.
- *(docs)* `ReportDrawnWhen` tells the system when the screen is really usable, so startup measures the right point.

- *(docs)* The docs also say to test a custom profile to confirm it helps, and to make sure ProfileInstaller 1.3 or later is in the target app.
- *(docs)* Generating on a Gradle Managed Device needs an `aosp` image; newer releases allow Android 13+ without root. Check the current page for your setup.
- *(general knowledge, not checked)* Include the profile in the **release** build; it is delivered to users with the app (check the Play delivery details in the docs).

Compose-Specific Performance, Revisited

- From Chapter 1: defer state reads with lambda modifiers, keep lazy-list keys stable, do not rely on recomposition counts alone *(docs)*.
- *(docs)* Composition tracing can be turned on in a Macrobenchmark with `androidx.benchmark.fullTracing.enable`, with a warning that tracing during cold start distorts the timing, so use it to find *what* is slow, and the untraced run to measure *how slow*.

Fixing Order

1. Turn on R8 and measure again *(Chapter 5; general knowledge about how big the win usually is: check on your app)*.
2. Run StrictMode in debug and fix what it reports.
3. Add a baseline profile and compare start-up.
4. Only then look at individual composables, and only the ones a trace points at.

> **COMPARED WITH THE CLASSIC UI COURSE**
>
> Android Development: Production & Publishing 6 teaches performance and optimisation: profiling, memory, ANRs and baseline profiles. The tools are the same; this edition leans on Macrobenchmark and the Compose drawn-state APIs, and ties R8 (Chapter 5) and the Compose habits (Chapter 1) into one measuring routine.

Try It Yourself

EXERCISE 1

Add the StrictMode block to the debug build and use the app normally. Report every violation it logs (it will probably find something). For each, say what it is and fix one.

View solution

EXERCISE 2

Create the leak from the chapter on purpose, rotate the screen a few times, take a heap dump in the Profiler and find the leaked Activity. Then fix it and repeat. Report how many Activity instances you saw before and after.

View solution

EXERCISE 3

Add a Macrobenchmark module with the template, run the two startup tests on a real device if you can, then generate a baseline profile and rerun. Report the numbers and the device, and say whether the profile helped. Do not assume the docs' figures apply.

View solution

NOT YET VERIFIED

Checked in the docs (three searches, summaries only): the baseline-profile recommendation, templates and generation; the profile scope beyond startup; `ReportDrawn` and friends; `FrameTimingMetric` and `testTagAsResourceId`; the version requirements as shown; profileable targets and ProfileInstaller; `StartupTimingMetric` and `CompilationMode` with the quoted example numbers; Gradle Managed Device notes; composition tracing and its caveat; the vitals ANR dashboard; the ANR causes; StrictMode thread and VM policies and `detectActivityLeaks()`; debug-only use; heap dumps and the LeakCanary profiler task.

General knowledge, not checked: what a leak is and the long-lived-object example; the leaky code as written; `penaltyLog` and `penaltyDeath`; "measure on a release-like build"; the explanation of why Compose benefits from profiles and ahead-of-time compilation; shipping the profile in the release build; the fixing order; the ViewModel remark; every import and the code as written.

Not verified: the ANR timeout figures; the Profiler's CPU and frame views; LeakCanary's own setup; the current library versions; every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install, and the docs' startup numbers are not yours.

Sources: Baseline Profiles overview (developer.android.com), Create Baseline Profiles (developer.android.com), Use a baseline profile, Compose (developer.android.com), Write a

Macrobenchmark (developer.android.com), Macrobenchmark instrumentation arguments (developer.android.com), Diagnose and fix ANRs (developer.android.com), StrictMode API reference (developer.android.com), App startup analysis and optimization (developer.android.com).

COMING UP

Chapter 7 is Publishing to Google Play: signing, app bundles, the Play Console, the store listing and release tracks, all described from the docs, since none of it can be done for you here.

Android Development with Compose: Production & Publishing

Course 6 · Chapter 7 · Going Live

This is the chapter most likely to be out of date by the time you read it: Play policy and console screens change often. It describes the route and what to prepare. It does **not** publish anything, create an account or touch a console, and nothing here was seen on a real Play Console. Treat the Play Console Help pages as the source of truth on the day.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked on 2026-11-22 (three searches, summaries only; the first search returned mostly third-party blogs, which are marked as such): Official pages (as summarised by the search): Signing: Android needs every APK signed; app bundles are signed with an upload key before upload and Play App Signing handles the rest; in Android Studio, Build > Generate Signed Bundle / APK creates the keystore and key; validity should be at least 25 years; the upload key and the app signing key should differ; a lost upload key can be reset through the Play Console; the Google-held signing key's fingerprint is what API providers need; once an app is on an open track or production its signing key is fixed. **New personal accounts:** the Play Console Help page says accounts created after 13 November 2023 must run a closed test with at least 12 testers opted in continuously for 14 days before applying for production access; a tester who leaves and rejoins restarts the count; until it is met, Production and Pre-registration stay disabled. (A community post said the figure was once 20 and was lowered to 12; the official page's 12 is the one used.) **Releases:** closed testing and creating a track are under Test and release > Testing; creating a release needs the "Release apps to testing tracks" permission. **Target API:** the Play Help page on target API levels, as summarised, says new apps and updates must target Android 16 (API level 36) or higher from 31 August 2026. **Blogs also mentioned an extension to 1 November 2026, which has already passed, so check what applies now. Third-party blogs only, not confirmed against an official page:** listing size limits (title 30 characters, short description 80, full 4,000, icon 512x512, feature graphic 1024x500), the Data safety form being required for every app, content rating and a privacy policy being required, an app bundle being required for new apps since 2021, a device-verification step for new personal accounts (an official Help page exists, but its contents were not read), a verified-developer requirement and a package-name registration deadline of 30 September 2026. **Not found:** versionCode rules, the bundle Gradle task, staged rollout percentages, what each testing track does, and the review process. **Everything else is general knowledge**, labelled so. **Code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen in any console.

Before You Start: What You Need

- **A Play developer account.** *(general knowledge)* Created by you, with a registration fee and identity checks that Google decides on. Not covered by the searches: look at the "Get started with Play Console" help page. Nothing here creates it for you.
- **A finished, tested release build** from Chapters 5 and 6 *(R8 on, tested by hand)*.
- **A privacy policy** at a public URL *(blogs say it is required: confirm)*, a store listing and screenshots.

Version and Target

```
// app/build.gradle.kts (general knowledge; check the current DSL for your AGP)
android {
    defaultConfig {
        applicationId = "com.yourname.notes"     // permanent once published: choose it
        minSdk = 26
        targetSdk = 36                           // docs (via a Play Help summary): new
        versionCode = 1                          // a whole number that must go up with
        versionName = "1.0.0"                    // what users see
    }
}
```

- *(general knowledge)* `versionCode` is a whole number that must increase with every upload, `versionName` is the label people read. The searches did not cover the rule; confirm in the docs.
- *(official page, as summarised)* New apps and updates must target API 36 or higher from 31 August 2026. That date has passed; the extension mentioned in blogs ended on 1 November 2026. **Check the live page: the target level rises about once a year.**
- *(general knowledge)* Raising `targetSdk` opts you in to newer behaviour changes, so test the app after raising it, in particular the permission and foreground-service changes of Chapters 3 and 4.

Signing

```
# In Android Studio: Build > Generate Signed Bundle / APK (docs), then:
#   1. choose Android App Bundle
```

```
#   2. create a NEW keystore and key (this is your UPLOAD key)
#   3. keep the keystore file and its passwords somewhere safe, OUTSIDE the repository
#   4. choose the release build variant and finish
#
# Docs (Android Studio / Play Help):
#   - bundles are signed with an upload key before upload; Play App Signing does the res
#   - the upload key and the app signing key should be different
#   - suggested validity: at least 25 years
#   - a lost upload key can be reset through the Play Console
#   - register the Google-held app signing key's fingerprint with API providers (Maps, O
#   - the signing key is fixed once the app is published to an open track or production
```

- **Why two keys.** *(docs)* You sign with an *upload key*; Google holds the *app signing key* that is much better than the old model, where losing the one key meant you could never update your app.
- **Back it up.** *(general knowledge)* Keep the keystore and its passwords out of the repository (Chapter 5) but safely stored somewhere you will still have in three years.
- **Fingerprints.** *(docs)* If you use Maps or Google sign-in, register the Google-held signing key's fingerprint, or those features may fail in the store version while working in your own builds.

```
# From the command line (general knowledge, not checked in these searches):
./gradlew bundleRelease
# The .aab appears under app/build/outputs/bundle/release/ (check the path on your machi
```

(blogs, unconfirmed) New apps are uploaded as an **Android App Bundle (.aab)** and not an APK. *(general knowledge)* Google Play then builds the right APKs for each device from the bundle.

The Play Console Route (described, not performed)

1. **Create the app** in the console: name, language, free or paid *(general knowledge; the "Create and set up your app" help page came up but was not read)*.
2. **Fill in the app content declarations**: privacy policy, Data safety form, content rating *(blogs say all are required; read the official pages)*. Answer from what the app really does. The Data

safety answers must match the real behaviour, including what Firebase (Chapter 8) collects *(general knowledge)*.

3. **Store listing**: title, descriptions, icon, graphics, screenshots *(blogs list the sizes above: confirm in the console, which tells you its limits)*.
4. **Upload a release** to a track *(docs: closed testing is under Test and release > Testing, and creating a release needs the testing-release permission)*.
5. **Test, then ask for production**, if your account needs the closed test (next section).
6. **Review and rollout** *(general knowledge, not in the searches)*: Google reviews the app; a staged rollout lets you release to a percentage of users first. Confirm the options in "Prepare and roll out a release".

Testing Tracks and the New-Account Rule

TRACK	PURPOSE *(GENERAL KNOWLEDGE; THE SEARCHES DID NOT DESCRIBE EACH)*
Internal testing	Quick checks by a small group, such as you and friends
Closed testing	A chosen list of testers; *(official page)* the one used for the new-account requirement
Open testing	Anyone can join; the app's signing key is fixed once it is on an open track *(docs)*
Production	Everyone, in the store

- *(official page)* A personal account created after 13 November 2023 needs a closed test with at least 12 testers opted in continuously for 14 days before applying for production. A tester who leaves and rejoins restarts the count, and Production stays disabled until the requirement is met.
- *(general knowledge)* Plan for this: you need 12 real people with Android phones who will stay opted in. Start the test early, and keep a list of who has accepted. Whether your account is covered depends on when it was created and on its type: the console tells you.

Before You Press Publish

```
Release checklist (the notes app)
[ ] applicationId chosen and final
[ ] targetSdk meets the CURRENT Play requirement (check the live Play Console Help page)
```

```
[ ] versionCode higher than any earlier upload; versionName set
[ ] R8 on for release (Chapter 5), release build tested by hand
[ ] No debug-only code or keys; StrictMode only in debug (Chapter 6)
[ ] Privacy policy URL ready; Data safety answers match what the app really does
[ ] Signed .aab built with the upload key; keystore and passwords backed up
[ ] Store listing text and graphics ready; screenshots from a real run
```

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Production & Publishing 7 teaches publishing to Google Play. The path is the same; what changes is that this edition leads with the current rules (a high target API level, Play App Signing with an upload key, the closed-test requirement for newer personal accounts) and says honestly which of them were confirmed on an official page and which only on blogs.

Try It Yourself

EXERCISE 1

Set `applicationId`, `versionCode` and `versionName`, and build a signed release bundle with a new keystore. Report where the keystore is stored, how you backed it up, and the path of the bundle. Do not upload anything.

View solution

EXERCISE 2

Open the Play Console Help pages for target API level and for the new-account testing requirement. Write down today's target level, whether your account is covered by the 12-tester rule, and anything that differs from this chapter.

View solution

EXERCISE 3

Work through the release checklist for the notes app and mark each item done or not. Write the Data safety answers you would give, based on what the app actually stores and sends. Add anything the checklist is missing.

View solution

NOT YET VERIFIED

Confirmed against official pages (as summarised by search results): signing and Play App Signing statements; the new-account closed-test rule; the track and permission pointers; the API 36 requirement and its date.

Blogs only, unconfirmed: listing limits; the Data safety, content rating and privacy-policy requirements; the bundle requirement; device verification; the verified-developer requirement and 30 September 2026 deadline.

General knowledge, not checked: the Gradle snippet and the `versionCode` rule; the bundle command and output path; the track purposes other than closed testing; review and staged rollout; the account fee and identity checks; the checklist.

Not verified: anything seen in a real console; the live state of any rule on the day you read this. Nothing was published, signed or uploaded by me.

Sources: Sign your app (developer.android.com), Use Play App Signing (Play Console Help), App testing requirements for new personal developer accounts (Play Console Help), Target API level requirements for Google Play apps (Play Console Help), Prepare and roll out a release (Play Console Help), Create and set up your app (Play Console Help), Set up an open, closed, or internal test (Play Console Help).

COMING UP

Chapter 8, the last, is Firebase: Crashlytics, Analytics, Remote Config and push messages, described from the docs, and the last of the three risky chapters.

Android Development with Compose: Production & Publishing

Course 6 · Chapter 8 · Hearing From Your App After Launch

Once an app is live you need to know when it crashes, how it is used, and be able to change something without shipping an update. Firebase covers those, and this last chapter adds four of its services to the notes app. Like Chapter 7, it describes the console steps and does not perform them: no Firebase project was created and nothing was seen in a console, and the setup details change often.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked on 2026-11-22 (five searches against the Firebase docs, summaries only): Setup: the Google services Gradle plugin goes in the project-level Gradle file and is applied in the app module; it processes `google-services.json`, which belongs in the app module directory; the BoM keeps library versions compatible but does not add libraries; library lines then omit versions; the config file name must not gain extras such as "(2)"; the config file's contents are considered public (package name, API key, project id); **the docs' own pages disagreed on the BoM version (35.2.0 in one place, 35.1.0 in another), and a Crashlytics page showed much older pinned versions**, so no version numbers are given here; the KTX modules for Crashlytics, Analytics, Messaging and Remote Config are deprecated or unsupported, so use the plain modules. **Crashlytics:** plugin in the project-level file then applied in the app file; `setCustomKey` keys apply to later fatal, non-fatal and ANR reports, with up to 64 pairs; `recordException` records a non-fatal in a catch block; only the most recent 8 recorded exceptions are stored; put unique values such as user ids in custom keys, not exception messages. **Remote Config:** libraries via the BoM; `setConfigSettingsAsync` with a minimum fetch interval (low only in development); in-app defaults from an XML resource through `setDefaultsAsync`; `fetchAndActivate()` in one call; real-time updates optional, on newer SDKs. **Messaging:** extend `FirebaseMessagingService` to handle foreground notifications and data messages; `onMessageReceived` runs on a background thread and should finish within 20 seconds; `onNewToken` fires when a token is first generated or changes, **but its reference page marks it deprecated in favour of** `onRegistered`; a notification-only message in the background goes to the system tray while a data message reaches the app (a table from the Unity guide, not the Android one); more than 100 pending messages or a month offline can trigger `onDeletedMessages()`; a newer AndroidConfigV2 API was labelled an early-access SDK. **Not found:** Analytics `logEvent` (general knowledge below), the Play Data safety link, the manifest entry for the messaging service, console screens, quotas and pricing. **Code is unrun** (no Kotlin compiler on the machine used to write this); nothing was seen in any console.

Setting Up (described, not performed)

1. **In the Firebase console** *(you do this)*: create a project, register your Android app with its `applicationId`, and download `google-services.json` *(docs)*. Keep the file name exact and put it in `app/`.
2. **In Gradle**: the plugin and the BoM, as below.
3. **Sync and run** once so the SDK initialises *(general knowledge)*.

```
// settings.gradle.kts / project-level build file: the Google services plugin (docs: pro
plugins {
    id("com.google.gms.google-services") version "<plugin-version>" apply false
    id("com.google.firebase.crashlytics") version "<crashlytics-plugin-version>" apply f
}

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

dependencies {
    // The BoM keeps Firebase library versions compatible; it does not add libraries its
    implementation(platform("com.google.firebase:firebase-bom:<bom-version>"))

    implementation("com.google.firebase:firebase-crashlytics")      // no -ktx: the docs
    implementation("com.google.firebase:firebase-analytics")
    implementation("com.google.firebase:firebase-config")
    implementation("com.google.firebase:firebase-messaging")
}
// Plugin and BoM versions change often: the docs' own pages disagreed with each other.
// Take the current numbers from the Firebase setup page, never from this listing.
// The config file google-services.json goes in the app/ module directory (docs).
```

- *(docs)* The config file is considered public, so do not treat it as a secret; but *(general knowledge)* how much a key can do is controlled by the restrictions you set in the console and the Firebase security rules, not by hiding the file. Check that in the docs for each product you use.

- *(general knowledge)* A release build needs the same setup; if you create a separate Firebase app for debug builds, give it its own `applicationId` suffix.

Crashlytics: Know When It Breaks

```
import com.google.firebase.crashlytics.FirebaseCrashlytics

// Context for a crash report: key-value pairs (docs: up to 64)
FirebaseCrashlytics.getInstance().setCustomKey("screen", "notes_list")

// A non-fatal problem you handled but want to know about (docs: call in a catch block)
try {
    repository.refresh()
} catch (e: Exception) {
    FirebaseCrashlytics.getInstance().recordException(e)
}

// Docs: only the most recent 8 recorded exceptions are stored, and unique values
// (user ids, timestamps) belong in custom keys, not exception messages.
// Privacy (general knowledge): never put note text or personal data in keys or messages
```

- *(docs)* Custom keys give context to every later report and can filter them in the console; non-fatals show problems you handled but want to count.
- *(general knowledge)* Crash traces from an R8 release build are obfuscated; the Crashlytics Gradle plugin uploads the mapping file so they can be read (confirm in the docs). That is why Chapter 5 said to keep `mapping.txt`.

Analytics: See How It Is Used

```
// General knowledge, NOT checked in these searches: logging an event
val analytics = Firebase.analytics        // import com.google.firebase.Firebase and com
analytics.logEvent("note_added") {
    param("length_bucket", "short")       // a coarse bucket, not the note text
}
```

(general knowledge, not checked in these searches) Analytics records screens and events you choose. Choose events that answer a question ("do people use pinned notes?"). Never send note text, and say what you collect in your privacy policy and in the Play Data safety form (Chapter 7).

Remote Config: Change It Without an Update

```
import com.google.firebase.Firebase
import com.google.firebase.remoteconfig.remoteConfig
import com.google.firebase.remoteconfig.remoteConfigSettings

class FeatureFlags {
    private val config = Firebase.remoteConfig

    init {
        config.setConfigSettingsAsync(remoteConfigSettings {
            minimumFetchIntervalInSeconds = 3600        // docs: use a low interval only
        })
        config.setDefaultsAsync(R.xml.remote_config_defaults)    // in-app defaults, use
    }

    fun refresh(onDone: () -> Unit) {
        config.fetchAndActivate().addOnCompleteListener { onDone() }     // fetch and ac
    }

    val showPinnedSection: Boolean get() = config.getBoolean("show_pinned_section")
}
```

```
<!-- res/xml/remote_config_defaults.xml (docs: defaults can live in an XML resource) -->
<defaults>
    <entry>
        <key>show_pinned_section</key>
        <value>false</value>
    </entry>
</defaults>
```

- *(docs)* Defaults in an XML resource are used until values have been fetched *and activated*; `fetchAndActivate()` does both in one call.
- *(docs)* A low minimum fetch interval is for development; *(general knowledge)* in production use a sensible interval, because fetches are rate-limited.
- *(general knowledge)* Use it for flags and tuning (a feature on or off, a limit), not for anything that must be secret or that would break the app if the value is wrong. Always have a safe default.

Cloud Messaging: Push from a Server

```
import com.google.firebase.messaging.FirebaseMessagingService
import com.google.firebase.messaging.RemoteMessage

class NotesMessagingService : FirebaseMessagingService() {

    // Docs: called for data messages, and for notification messages while the app is in
    // runs on a background thread; finish within about 20 seconds
    override fun onMessageReceived(message: RemoteMessage) {
        val text = message.data["text"] ?: message.notification?.body ?: return
        showReminder(applicationContext, noteId = 0L, text = text)       // Chapter 4's
    }

    // Docs: called when a token is first generated and when it changes.
    // The reference page marks onNewToken deprecated in favour of onRegistered: check t
    override fun onNewToken(token: String) {
        // send the token to YOUR server so it can address this device (never log it in
    }
}

// Manifest: register the service with the MESSAGING_EVENT intent filter (general knowle
// Android 13+: showing a notification still needs POST_NOTIFICATIONS: Chapter 4's permi
```

- *(docs)* Extending `FirebaseMessagingService` is needed for foreground notifications and for data payloads. The callbacks run on a background thread; `onMessageReceived` should finish within 20 seconds.
- *(docs)* A notification-only message sent while the app is in the background goes to the system tray, and a data message is passed to your app; that table was in the Unity guide, so

check the Android page for the Android behaviour.

- *(docs)* `onNewToken` reports a new or changed token. The reference page marks it deprecated in favour of `onRegistered`, so read the current guide before copying this method.
- *(docs)* With many pending messages or a long time offline you may get `onDeletedMessages()`, and the app should do a full sync with its server.
- *(general knowledge)* You need a server (or the console's message composer) to send. The notes app would use it for "a note was changed on another device", and then trigger Chapter 3's sync. Showing a notification still needs the permission flow of Chapter 4.

What to Tell Your Users

SERVICE	COLLECTS *(GENERAL KNOWLEDGE, CHECK EACH PRODUCT'S DOCS)*	DISCLOSE IN
Crashlytics	Crash and device information	Privacy policy; Data safety form
Analytics	Events and device identifiers	Privacy policy; Data safety form; consent where required
Remote Config	Fetch requests	Privacy policy
Cloud Messaging	A device token	Privacy policy; Data safety form

Adding these changes your answers to Chapter 7's Data safety form: do it before you publish, and again whenever you add a service.

COMPARED WITH THE CLASSIC UI COURSE

Android Development: Production & Publishing 8 teaches Firebase: Crashlytics, Analytics, Remote Config and push messages. The services are the same; this edition says that the KTX modules are no longer supported, that version numbers in the docs disagreed with each other so none are copied here, and that `onNewToken` is marked deprecated, all as seen in the docs on 2026-11-22.

Try It Yourself

EXERCISE 1

Create a Firebase project and add the notes app, using the current setup page and not the listing here. Add Crashlytics, then cause a deliberate test crash in a debug build and find it in the console. Report the versions the setup page gave you and any step that differed from this chapter.

View solution

EXERCISE 2

Add Remote Config with the `show_pinned_section` flag. Change it in the console, fetch it in the app and show or hide a section. Report how long it took to see the change and what the fetch interval did.

View solution

EXERCISE 3

Add `NotesMessagingService`, send a test message from the console to your device in both the foreground and the background, and report which callbacks ran. Then read the current guide on `onNewToken` against `onRegistered` and update the code. Finally, update your Data safety answers from Chapter 7 for the services you added.

View solution

NOT YET VERIFIED

Checked in the docs on 2026-11-22 (five searches, summaries only): the plugin, the config file and the BoM roles; the KTX deprecation; Crashlytics custom keys, non-fatals, the 8-exception and 64-key limits; Remote Config settings, defaults and `fetchAndActivate`; the messaging service, its threading and 20-second limit; `onNewToken`'s deprecation note; `onDeletedMessages`.

General knowledge, not checked: Analytics `logEvent` and its imports; mapping-file upload by the Crashlytics plugin; the manifest entry for the messaging service; the safety of the config file depending on restrictions and rules; Remote Config rate limits and sensible use; the data-collection table; every import and the code as written.

Not verified: any version number (the docs disagreed with themselves); Android-specific background delivery behaviour (the table was from the Unity guide); console screens, quotas and pricing; every listing and solution (unrun); the exercises (not run by me). Nothing was created in Firebase or the Play Console by me. **Re-check the live Firebase docs before you start.**

Sources: Add Firebase to your Android project (firebase.google.com), Google services Gradle plugin and JSON config file (firebase.google.com), Get started with Firebase Crashlytics (firebase.google.com), Customize crash reports for Android (firebase.google.com), Get started with Remote Config on

Android (firebase.google.com), Get started with Firebase Cloud Messaging in Android apps (firebase.google.com), Receive messages in Android apps (firebase.google.com), FirebaseMessagingService API reference (firebase.google.com).

COURSE COMPLETE

That is the end of Android Development with Compose: Production & Publishing, and the end of the Compose-first path: Android Development Fundamentals with Compose, Android Development with Compose: Architecture & Data, and this course. The notes app is now tested, backgrounded, notifying, secured, measured and ready to publish, as far as a course can take it without a console of your own. The catalogue project in the Android section is the natural next build.