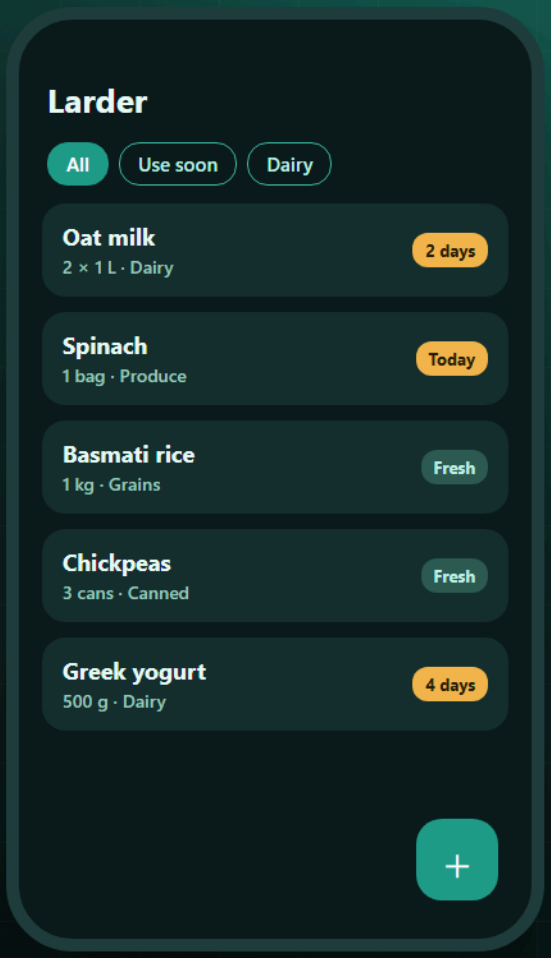
```
// Jetpack Compose
@Composable
fun PantryList(
  items: List<Item>
) {
  LazyColumn {
    items(items) {
      ItemCard(it)
    }
  }
}

// ViewModel + Flow
uiState.collectAs…
StateWithLifecycle()
```



THE COMPLETE PRACTICAL GUIDE

Android Development with Kotlin

Build Modern Android Apps with Jetpack Compose

Kotlin · Compose · Material 3 · ViewModel · Hilt · Room · Retrofit · Offline-First · Testing · Google Play

28 CHAPTERS · 4 APPENDICES · ONE COMPLETE APP

THENEXBOOKS

Android Development with Kotlin

The Complete Practical Guide to Building Modern Android Apps with Jetpack Compose

From Your First Composable to the Google Play Store

First Edition

THENEXBOOKS

Android Development with Kotlin
The Complete Practical Guide to Building Modern Android Apps with Jetpack Compose

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Android, its libraries, tools, and Google Play policies change frequently; always consult the official documentation for the versions you use.

Android, Google Play, Jetpack Compose, and Android Studio are trademarks of Google LLC. Kotlin is a trademark of the Kotlin Foundation. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with or endorsed by Google or JetBrains.

Larder, the example app in this book, is fictional. Any resemblance to real apps or companies is coincidental.

Preface

Android runs on an enormous range of devices — phones, tablets, foldables, TVs, cars, and watches — and building for it has changed more in the last few years than in the decade before. Kotlin has replaced Java as the preferred language. Jetpack Compose has replaced XML layouts as the recommended way to build user interfaces. Coroutines and Flow have replaced callbacks. A clear, recommended architecture has replaced a patchwork of patterns.

That is good news for anyone starting today. Modern Android development is more concise, more consistent, and more enjoyable than it has ever been. But it also means that many tutorials and answers online describe an older way of working, and it can be hard to know which parts still matter.

This book is a complete, practical guide to building Android apps the modern way. We start with Kotlin itself, build user interfaces with Compose, structure the app with ViewModels, Hilt, and unidirectional data flow, store data with Room and DataStore, talk to the network, work offline, run background work, use the camera, test, optimize, adapt to large screens, and finally ship to Google Play. Every chapter builds on one realistic app, so you will see how the pieces fit together.

Who This Book Is For

- **Developers new to Android** — web, backend, or Java developers who want to build native mobile apps.
- **Students and career changers** looking for a structured path into mobile development.
- **Android developers** moving from XML Views and older patterns to Compose and modern architecture.
- **Kotlin developers** who want to apply the language to real Android apps.

You should be comfortable with basic programming concepts. No prior Kotlin or Android experience is required.

What You Will Learn

- Kotlin from the essentials to idiomatic features, plus coroutines and Flow.
- Jetpack Compose: layouts, state, lists, Material 3 theming, navigation, forms, and accessibility.
- Modern app architecture with ViewModels, unidirectional data flow, Hilt, and lifecycle awareness.
- Gradle, build variants, version catalogs, and modularization.
- Room, DataStore, Retrofit with kotlinx.serialization, and offline-first repositories.
- WorkManager, notifications, permissions, the camera, and photos.
- Testing, performance, adaptive layouts, security, and privacy.
- Signing, app bundles, R8, the Play Console, and life after launch.

How This Book Is Organized

Part I — Kotlin for Android (Chapters 1–5) sets up your tools and teaches Kotlin and coroutines. **Part II — Building UIs with Jetpack Compose** (Chapters 6–11) covers everything you need to build screens. **Part III — App Architecture** (Chapters 12–15) structures the app for growth. **Part IV — Data and Networking** (Chapters 16–20) adds persistence, networking, offline support, and background work. **Part V — Device Features and Quality** (Chapters 21–25) covers the camera, testing, performance, large screens, and security. **Part VI — Shipping** (Chapters 26–28) brings the app together and releases it. The appendices provide cheat sheets, a release checklist, and a glossary. Each chapter ends with a **Key Takeaways** box.

The Running Example: Larder

Throughout the book we build **Larder**, a fictional pantry and meal-planning app. You will track pantry items and their expiry dates, browse recipes from an online API, plan meals for the week, generate a shopping list, get reminders before food goes to waste, capture items with the camera, and use the app offline — on phones, tablets, and foldables.

Conventions Used in This Book

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version details, or a useful aside.

> **WARNING**
>
> A common mistake that causes crashes, data loss, or policy problems.

Code appears in `monospace`, with the file path above each example. Library versions change quickly, so version numbers in build files are examples — always use the latest stable versions recommended in the official documentation.

Contents

Preface

PART I — KOTLIN FOR ANDROID

1 Android Development Today: The Platform, the Tools, and What We Will Build

2 Setting Up Android Studio and Running Your First App

3 Kotlin Essentials: Variables, Types, Functions, Control Flow, and Null Safety

4 Kotlin in Depth: Classes, Collections, Lambdas, Extensions, and Scope Functions

5 Coroutines and Flow Fundamentals

PART II — BUILDING UIS WITH JETPACK COMPOSE

6 Compose Fundamentals: Composables, Modifiers, and Layouts

7 State, Recomposition, and State Hoisting

8 Lists, Lazy Layouts, and Images

9 Material 3 Design, Theming, Typography, and Dark Mode

10 Navigation in Compose

11 Forms, Input, Dialogs, and Accessibility

PART III — APP ARCHITECTURE

12 Modern App Architecture: UI Layer, ViewModel, and Unidirectional Data Flow

13 Dependency Injection with Hilt

14 Lifecycle, Configuration Changes, and Process Death

15 Gradle, Build Variants, Version Catalogs, and Modularization

PART IV — DATA AND NETWORKING

16 Local Data with Room

17 Settings and Small Data with DataStore

18 Networking with Retrofit and kotlinx.serialization

19 Offline-First Apps: Repositories, Caching, and Sync

20 Background Work with WorkManager, Notifications, and Permissions

PART V — DEVICE FEATURES AND QUALITY

21 Camera, Photos, Location, and Runtime Permissions

22 Testing Android Apps: Unit, Compose UI, and Integration Tests

23 Performance: Startup, Baseline Profiles, Memory, and Smooth UI

24 Adaptive Layouts for Tablets, Foldables, and Large Screens

25 Security and Privacy Best Practices

PART VI — SHIPPING

26 Case Study: Building Larder End to End

27 Preparing for Release: Signing, App Bundles, R8, and the Play Console

28 After Launch: Crash Reporting, Analytics, Updates, and Where to Go Next

APPENDICES

A Kotlin Cheat Sheet

B Compose and Architecture Reference

C Release Checklist

D Glossary

PART I

Kotlin for Android

The platform, the tools, and the language: set up Android Studio, run your first app, and learn the Kotlin — including coroutines and Flow — that every modern Android app is written in.

1 · Android Development Today
2 · Android Studio and Your First App
3 · Kotlin Essentials
4 · Kotlin in Depth
5 · Coroutines and Flow

CHAPTER 1 · PART I — KOTLIN FOR ANDROID

Android Development Today: The Platform, the Tools, and What We Will Build

Before we write a single line of code, it helps to know what Android actually is, how a modern app is put together, and which tools you will live in every day. This chapter gives you that map, introduces Larder, the app we will build together across the book, and explains how the rest of the chapters fit around it.

Why Android, and Why Now

Android runs on an enormous range of hardware: phones at every price point, tablets, foldables, watches, televisions, car dashboards, and a long tail of specialized devices such as point-of-sale terminals and handheld scanners. For a developer, that reach is the main attraction. An app you build with the techniques in this book can run on devices you will never personally hold, used by people in countries you may never visit.

The way Android apps are written has also changed a great deal in recent years. If you looked at Android development a while ago, you may remember Java code, XML layout files, `findViewById` calls, fragments with complicated lifecycles, and `AsyncTask` for background work. Most of that has been replaced. Today the recommended stack looks like this:

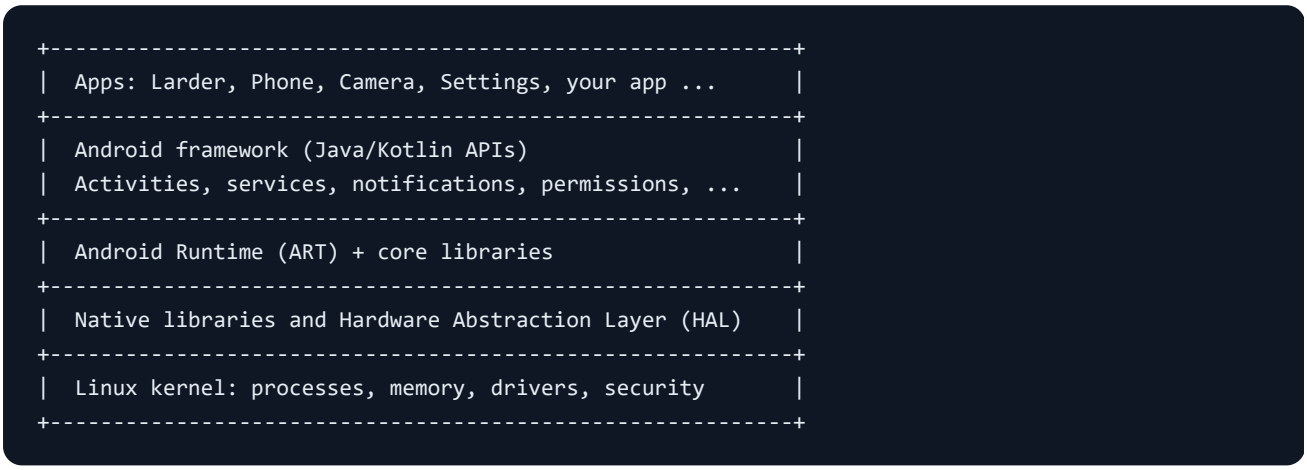
- **Kotlin** as the programming language. Google has declared Kotlin its preferred language for Android, and new documentation, samples, and libraries are written Kotlin-first.
- **Jetpack Compose** as the UI toolkit. Compose is the recommended modern way to build Android user interfaces. You describe your screens with Kotlin functions instead of XML files.
- **Coroutines and Flow** for asynchronous work and streams of data, replacing callbacks and older threading utilities.
- **Jetpack libraries** such as ViewModel, Room, DataStore, Navigation, WorkManager, and Hilt for architecture, storage, navigation, background work, and dependency injection.
- **Gradle with version catalogs** for building, with Android Studio as the development environment.

This combination is sometimes called "Modern Android Development". It is the stack this book teaches from the first page. We will not spend time on the older approaches except where you are likely to meet them in existing code, in which case we will point them out so you recognize them.

The good news for newcomers is that the modern stack is considerably easier to learn than the old one. Compose removes a whole category of bugs caused by keeping XML views and Kotlin state in sync. Coroutines let you write asynchronous code that reads top to bottom. Kotlin's null safety catches mistakes at compile time that used to crash apps in production.

What Android Is, Layer by Layer

Android is an operating system built on top of the Linux kernel, plus a large framework that apps talk to. You rarely interact with the lower layers directly, but understanding their shape explains many rules you will meet later, such as why apps need permissions, why your process can be killed at any time, and why the main thread matters so much.



The Linux kernel and the sandbox

At the bottom is the Linux kernel, which manages processes, memory, drivers, and power. Android uses it to give every app its own Linux user ID and its own process. That is the foundation of the **application sandbox**: by default, your app cannot read another app's files or memory, and other apps cannot read yours. When your app wants something outside its sandbox, such as the camera, precise location, or the ability to post notifications, it has to ask for a permission, and for sensitive permissions the user has to grant it at runtime. We will handle runtime permissions properly in Chapter 20 and Chapter 21.

The Android Runtime

Your Kotlin code is compiled to JVM bytecode and then converted into DEX (Dalvik Executable) format, which is what the Android Runtime, ART, executes. ART compiles code ahead of time and just in time, using profiles to decide which methods deserve optimization. You do not need to understand ART's internals to build apps, but it explains why Android has its own tools for shrinking and optimizing code (R8, covered in Chapter 27) and why baseline profiles can make startup faster (Chapter 23).

The framework

The framework is the huge set of APIs your app calls: `Activity`, `Context`, `Intent`, `NotificationManager`, `PackageManager`, and many more. These APIs are part of the operating system on the device. That is an important detail: when you call a framework API, the implementation that runs is the one shipped with the user's Android version, not one bundled in your app. This is why newer framework APIs are only available on newer devices, and why Google moved much new functionality into the Jetpack libraries, which ship inside your app and work consistently across many Android versions.

API Levels and Why They Matter

Every Android release has an **API level**, an integer that identifies the version of the framework APIs. For example, Android 13 corresponds to API level 33. When a new feature appears, the documentation tells you which API level introduced it. Android 13, for instance, introduced the `POST_NOTIFICATIONS` runtime permission, so apps running on those devices must ask the user before posting notifications.

Your project declares three related numbers in its Gradle build file:

Setting	Meaning	Practical effect
`minSdk`	The oldest API level your app supports	Devices below it cannot install the app
`compileSdk`	The API level your code is compiled against	Which framework APIs you can reference
`targetSdk`	The API level you have tested and adapted to	Opts the app into newer platform behavior

Choosing `minSdk` is a trade-off between reach and convenience. A lower value lets more devices install the app but forces you to check the API level before using newer features. Jetpack libraries soften this by providing backward-compatible wrappers. Choosing `targetSdk` is less of a choice than it looks: Google Play requires apps to target a recent API level, and the requirement moves forward over time. Always check the current requirement in the Play Console documentation rather than relying on a number you remember.

When you do need to use an API that only exists on newer devices, you guard it with a version check:

app/src/main/java/com/example/larder/notifications/NotificationSupport.kt

```
package com.example.larder.notifications

import android.os.Build

fun needsNotificationPermission(): Boolean =
    Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU // Android 13, API 33
```

Android Studio's lint checks will warn you when you call an API that is newer than your `minSdk` without such a guard, so you do not have to memorize which API arrived when.

The Building Blocks of an Android App

An Android app is not a single `main` function that runs from start to finish. Instead, it is a set of **components** that the system can start independently. Each one is declared in the app's manifest file, `AndroidManifest.xml`, so the system knows it exists.

- **Activities** represent a screen or window the user interacts with. In a Compose app you usually have a single activity that hosts all of your composable screens, with navigation handled inside Compose.
- **Services** perform work without a user interface. Modern apps rarely write raw services; they use WorkManager for deferrable background tasks or foreground services for ongoing, user-visible work such as music playback.
- **Broadcast receivers** respond to system-wide events such as the device finishing boot or the time zone changing.
- **Content providers** share structured data between apps. You will use existing ones, such as the media store, more often than you write your own.

Components are started with **intents**, which are messages describing what should happen: "open this activity", "share this text", "view this web page". Intents are how apps cooperate. When Larder later lets the user share a shopping list, it will send an intent that any messaging or notes app can handle.

Here is a minimal manifest for a single-activity Compose app. Notice that XML inside this book is always shown with its angle brackets intact, exactly as you would type it in the file.

app/src/main/AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">

    <application
        android:name=".LarderApplication"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/Theme.Larder">

        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

The intent filter with the `MAIN` action and `LAUNCHER` category is what makes the activity appear as an icon in the device's app launcher. The `android:exported="true"` attribute says that other apps (in this case the launcher) are allowed to start it. Recent Android versions require you to set `exported`

explicitly on any component that has an intent filter.

The system is in charge of your process

One idea surprises almost every developer new to Android: the system, not your app, decides when your process lives and dies. When the user leaves your app and the device needs memory, Android may kill your process without warning. When the user comes back, the system recreates your activity and expects you to restore the screen the user left. Configuration changes, such as rotating the device, switching to dark mode, or changing the language, also recreate the activity by default.

This is why Android architecture puts so much emphasis on keeping state outside of UI objects, in places like `ViewModel` and persistent storage, and on saving small amounts of UI state that must survive process death. Chapter 14 is devoted to this topic. For now, just remember that an Android app must be ready to be stopped and restarted at any moment, and that this is normal behavior, not an error.

> **NOTE**
> The main thread (also called the UI thread) draws your interface and handles touch input. If you block it with slow work such as network calls or large database queries, the app freezes, and if it stays frozen long enough the system shows an "App Not Responding" dialog. Kotlin coroutines, introduced in Chapter 5, are the standard way to move work off the main thread without making your code hard to read.

Kotlin: The Language of Modern Android

Kotlin is a statically typed language created by JetBrains that runs on the JVM and interoperates fully with Java. That interoperability matters: every Java library and every framework API is callable from Kotlin, so moving to Kotlin never meant throwing away the existing ecosystem.

What makes Kotlin particularly well suited to Android is a handful of features you will use constantly:

- **Null safety.** The type system distinguishes between values that can be null and values that cannot, which removes most null pointer crashes at compile time.
- **Concise data modeling.** Data classes give you equality, copying, and readable `toString` output in one line.
- **Lambdas and higher-order functions.** Compose is built on functions that take other functions as parameters, and Kotlin's lambda syntax makes that natural.
- **Coroutines.** Asynchronous code written in a sequential style, with structured cancellation tied to lifecycles.
- **Extension functions.** You can add helpers to existing types, which Jetpack libraries use heavily.

Here is a first taste of Kotlin, modeling a pantry item for Larder. Do not worry about every keyword; Chapter 3 and Chapter 4 cover the language in detail.

app/src/main/java/com/example/larder/model/PantryItem.kt

```
package com.example.larder.model

import java.time.LocalDate

enum class Category { PRODUCE, DAIRY, MEAT, BAKERY, PANTRY, FROZEN, OTHER }

data class PantryItem(
    val id: Long,
    val name: String,
    val quantity: Double,
    val unit: String,
    val category: Category,
    val expiresOn: LocalDate? = null, // null means "does not expire"
)

fun PantryItem.isExpiringWithin(days: Long, today: LocalDate = LocalDate.now()): Boolean {
    val expiry = expiresOn ?: return false
    return !expiry.isAfter(today.plusDays(days))
}
```

In a few lines we have an immutable data type, an enumeration of categories, a nullable expiry date with a default value, and an extension function that answers the question the "use soon" list will ask. The `?:` operator, called the Elvis operator, returns early when the item has no expiry date. That kind of expressive, safe code is typical of Kotlin.

Jetpack Compose: Declarative UI

For most of Android's history, user interfaces were built with the **View system**: you declared a tree of widgets in XML, inflated it at runtime, looked up individual views, and then mutated them from code whenever data changed. The trouble with that approach is keeping two representations in sync. The XML defines the initial state, the code changes it piece by piece, and every missed update becomes a bug: a counter that shows a stale number, a button that stays enabled when it should not be, a list that does not refresh.

Compose takes a different approach, called **declarative UI**. You write functions, marked with `@Composable`, that describe what the screen should look like for the current state. When the state changes, Compose calls the affected functions again and updates only the parts of the screen that actually changed. This process is called **recomposition**. You never reach into a widget and set its text; you change the state, and the UI follows.

Here is a small composable that shows one pantry item as a row:

app/src/main/java/com/example/larder/ui/PantryItemRow.kt

```
package com.example.larder.ui

import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
```

```
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import com.example.larder.model.PantryItem

@Composable
fun PantryItemRow(item: PantryItem, modifier: Modifier = Modifier) {
    Row(modifier = modifier.fillMaxWidth().padding(16.dp)) {
        Column(modifier = Modifier.weight(1f)) {
            Text(text = item.name, style = MaterialTheme.typography.titleMedium)
            Text(
                text = item.category.name.lowercase().replaceFirstChar { it.uppercase() },
                style = MaterialTheme.typography.bodySmall,
            )
        }
        Text(text = "${item.quantity} ${item.unit}")
    }
}
```

On screen, this produces a full-width row with the item name in a medium title style, the category in small text beneath it, and the quantity with its unit aligned to the right. The function takes data in and emits UI; it holds no references to views and returns nothing. If the item's quantity changes, Compose calls `PantryItemRow` again with the new item and updates the text.

Compose still runs on top of the same Android platform. It draws into a regular activity, it respects the same lifecycle, and it can host traditional views when needed (for example, a map or an ad view that only exists as a `View`). Existing apps can adopt Compose one screen at a time. But for new apps and new screens, Compose is the toolkit Google recommends, and it is the toolkit we use throughout this book.

> **TIP**
>
> If you come from web development, Compose will feel familiar: composables are similar to components in declarative web frameworks, state flows down as parameters, and events flow up as callbacks. The main differences are that Compose is plain Kotlin (no templates, no separate markup language) and that layout is done with Kotlin functions such as `Row`, `Column`, and `Box` rather than a style sheet.

How a Modern App Is Structured

A small demo can live in one file. A real app with a database, a network API, background work, and many screens needs structure, or it becomes hard to change. Google's recommended architecture, which we cover in depth in Part III, divides an app into layers with clear responsibilities:

```
+-----------------------------+
|  UI layer                   |   Composables + ViewModels
|  (screens, UI state)        |   observe state, send events
+--------------+--------------+
               |
               v
+-----------------------------+
|  Domain layer (optional)    |   Reusable business rules
|  (use cases)                |   e.g. "build shopping list"
+--------------+--------------+
               |
               v
+-----------------------------+
|  Data layer                 |   Repositories combine
|  (repositories, sources)    |   Room, DataStore, network
+-----------------------------+
```

The **UI layer** contains composables that render state and `ViewModel` classes that hold that state and survive configuration changes. The **data layer** contains repositories that expose data to the rest of the app and hide where it comes from, whether a local Room database, DataStore preferences, or a remote REST API. An optional **domain layer** holds business logic that several screens share. Data flows in one direction, from the data layer up to the UI as observable state, and events flow the other way, from the UI down to the ViewModel. This pattern is called **unidirectional data flow**, and it makes screens predictable and testable.

To give you a feel for how these pieces connect, here is a preview of a ViewModel that will appear, in a more complete form, much later in the book:

feature/pantry/PantryViewModel.kt (preview)

```
class PantryViewModel(
    private val repository: PantryRepository,
) : ViewModel() {

    val uiState: StateFlow<PantryUiState> = repository.observeItems()
        .map { items -> PantryUiState(items = items.sortedBy { it.name }) }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5_000),
            initialValue = PantryUiState(isLoading = true),
        )

    fun onDeleteItem(id: Long) {
        viewModelScope.launch { repository.delete(id) }
    }
}
```

Almost every line of this snippet uses a concept we have not introduced yet: `StateFlow`, `map`, `stateIn`, `viewModelScope`, `launch`. That is intentional. By Chapter 12 each piece will be familiar, and you will be able to read and write code like this without thinking twice. Keep this snippet in mind as a destination.

The Tools You Will Use Every Day

Android Studio

Android Studio is the official IDE for Android development, built on JetBrains' IntelliJ platform. It includes a Kotlin-aware editor with refactoring tools, a Compose preview pane that renders composables without running the app, a layout inspector for examining the live UI tree, a device manager for emulators, a debugger, profilers for CPU, memory, and energy, and a database inspector for looking inside Room databases on a running device. Chapter 2 walks through installing and configuring it.

Gradle and the Android Gradle Plugin

Android projects are built with Gradle, a general-purpose build tool, extended by the Android Gradle Plugin (AGP), which knows how to compile Kotlin, process resources, merge manifests, package an app, and sign it. Build scripts are usually written in the Kotlin DSL, in files named `build.gradle.kts`, and library versions are centralized in a **version catalog** file called `libs.versions.toml`. You will see a first build file in Chapter 2 and learn Gradle properly in Chapter 15.

The emulator and physical devices

The Android Emulator runs virtual devices on your computer, with configurable screen sizes, API levels, and hardware features. It is invaluable for testing on configurations you do not own, such as a tablet or a foldable. A physical device, connected by USB or over Wi-Fi, gives you real performance characteristics and real sensors. Most developers use both.

adb

The Android Debug Bridge, `adb`, is a command-line tool that talks to emulators and connected devices. Android Studio uses it behind the scenes, but it is useful on its own for installing builds, reading logs, and running shell commands on a device:

```
adb devices                       # list connected devices and emulators
adb install app-debug.apk         # install a build
adb logcat -s Larder              # show log lines tagged "Larder"
adb shell am force-stop com.example.larder   # stop the app process
```

The last command is a handy way to simulate the system killing your process, which you will use when testing state restoration in Chapter 14.

Logcat

Logcat is Android's system log, visible in a dedicated Android Studio window. Your app writes to it with the `Log` class.

```
import android.util.Log

private const val TAG = "Larder"

fun logItemAdded(name: String) {
    Log.d(TAG, "Added pantry item: $name")
}
```

Jetpack: The Libraries Around the Platform

Jetpack is the umbrella name for Google's Android libraries that ship with your app rather than with the operating system. Because they are bundled, they behave the same on old and new devices and can be updated on their own schedule, independent of Android releases. Here are the ones we will use for Larder and where each is covered:

Library	What it does	Chapter
Compose UI and Material 3	Declarative UI and design components	6 to 11
Lifecycle and ViewModel	Screen state that survives configuration changes	12, 14
Navigation Compose	Moving between screens, back stack	10
Hilt	Dependency injection	13
Room	SQLite database with a type-safe API	16
DataStore	Settings and small key-value data	17
WorkManager	Reliable background work	20
CameraX	Camera preview and capture	21

Outside Jetpack, we will also use a few widely adopted open-source libraries: Retrofit and kotlinx.serialization for networking (Chapter 18) and an image loading library such as Coil (Chapter 8). The book always tells you how to add a dependency through the version catalog, and it never pins you to a specific version number in prose. Libraries evolve quickly; use the latest stable release when you start, and read the release notes when you upgrade.

> **WARNING**
>
> Online tutorials age quickly in the Android world. When you search for help, check the date and look for signs of an older approach: XML layouts for new screens, `AsyncTask`, `LiveData` used where `StateFlow` would be expected, or Gradle files in Groovy syntax with hard-coded versions. Older answers are not always wrong, but you should translate them to the modern stack rather than copying them directly into a Compose app.

Meet Larder, the App We Will Build

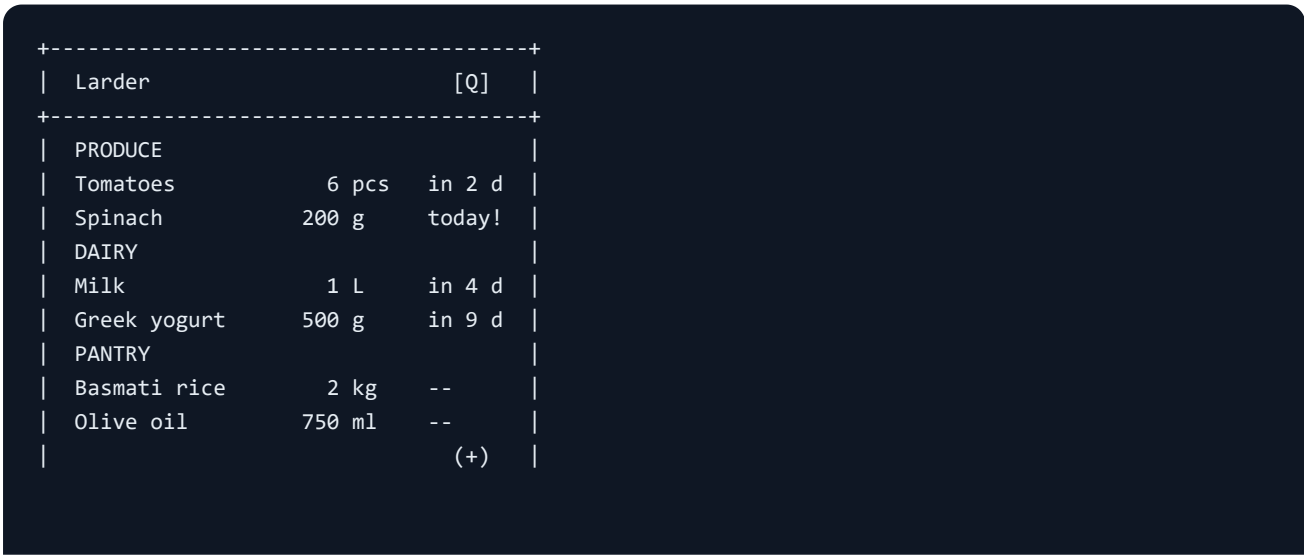
Learning from disconnected snippets makes it hard to see how the pieces fit together. So instead, this book follows one app from an empty project to a release on Google Play. That app is **Larder**, a pantry and meal-planning app. It is fictional, but its problems are real: storing structured data, showing lists, editing forms, fetching data from a server, working offline, running background tasks, and adapting to different screens.

What Larder does

- **Pantry.** A list of the food you have at home, with name, quantity, unit, category, and expiry date. You can add, edit, and delete items, search by name, and filter by category.
- **Use soon.** A focused list of items that expire in the next few days, so less food is wasted.
- **Recipes.** Recipes fetched from a fictional REST API at `https://api.larder.example/v1/`, with the ability to see which ingredients you already have.
- **Meal plan.** A weekly plan where you assign recipes to days.
- **Shopping list.** Generated automatically from ingredients your planned meals need but your pantry lacks.
- **Settings.** Theme and unit preferences stored with DataStore.
- **Reminders.** A daily background check with WorkManager that posts a notification when items are about to expire.
- **Capture.** Adding items by photo or barcode with the camera.
- **Offline-first sync, adaptive layouts, and release.** Larder works without a connection, looks good on tablets and foldables, and ships as an Android App Bundle.

A sketch of the main screen

Here is roughly what the pantry screen will look like once the core chapters are done. It has a top app bar with a search action, a scrollable list of items grouped by category, a floating action button for adding an item, and a bottom navigation bar for the main sections.



```
+-------------------------------------+
| Pantry | Use soon | Recipes | Plan   |
+-------------------------------------+
```

How Larder's code will be organized

Larder starts as a single `:app` module with the package `com.example.larder`. As it grows, Chapter 15 splits it into modules so that features can be built and tested independently and so that builds stay fast:

```
larder/
|-- app/                # :app - entry point, navigation, DI setup
|-- core/
|   |-- data/           # :core:data - repositories
|   |-- database/       # :core:database - Room
|   |-- network/        # :core:network - Retrofit client
|   `-- designsystem/   # :core:designsystem - theme, shared components
`-- feature/
    |-- pantry/         # :feature:pantry
    |-- recipes/        # :feature:recipes
    `-- mealplan/       # :feature:mealplan
```

You do not need to create this structure now. It is shown here so you can see where the book is heading: a codebase that looks like a professional project, not a toy.

A first glimpse of the entry point

Every Compose app starts in an activity that calls `setContent`. Larder's entry point will stay remarkably small for the whole book, because all the interesting work happens in composables, ViewModels, and repositories:

app/src/main/java/com/example/larder/MainActivity.kt

```
package com.example.larder

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import com.example.larder.ui.LarderApp
import com.example.larder.ui.theme.LarderTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            LarderTheme {
                LarderApp()
            }
```