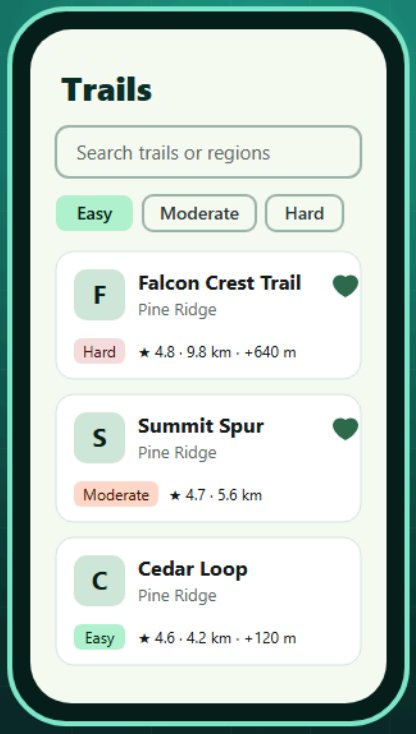
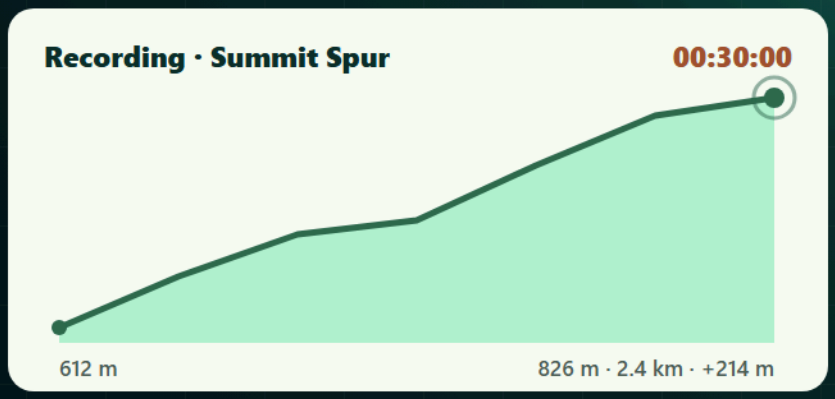
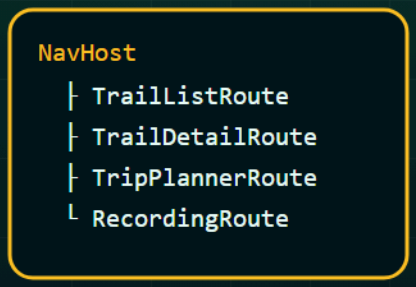
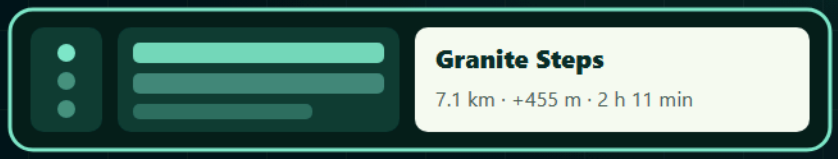










INTERMEDIATE TO ADVANCED

Jetpack Compose in Depth

Layouts, State, Material 3 Theming, Navigation, Animation, Accessibility, and Performance

Recomposition · Custom Layouts · Modifier.Node · Adaptive UI
Type-Safe Navigation · Motion · Semantics · UI Tests

KOTLIN · MATERIAL 3 · PHONES, TABLETS, FOLDABLES

THENEXBOOKS

Jetpack Compose in Depth

Layouts, State, Material 3 Theming, Navigation, Animation, Accessibility, and Performance for Real Android Apps

How Compose really works, and how to build polished, adaptive, accessible apps with it

First Edition

THENEXBOOKS

Jetpack Compose in Depth
Layouts, State, Material 3 Theming, Navigation, Animation, Accessibility, and Performance for Real Android Apps

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, Jetpack libraries, Kotlin, and Android Studio change frequently; APIs, default behaviors, tool windows, and menu names can differ between releases. Always consult the current official documentation for the versions you use.

Android, Jetpack Compose, Android Studio, Google Play, and Material Design are trademarks of Google LLC. Kotlin is a trademark of the Kotlin Foundation. Java is a registered trademark of Oracle and/or its affiliates. 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 independent and is not affiliated with, sponsored by, or endorsed by Google LLC, JetBrains, or the Kotlin Foundation. Screen sketches in this book are original illustrations.

Northbeam Labs, the Waypoint app, its team members, its trails, regions, trips, journal entries, users, and web addresses are fictional and are used only as a running example. Any resemblance to real companies, places, or persons is coincidental. The `.example` domains used in the book are reserved for documentation and do not belong to a real service.

Preface

Writing a first Jetpack Compose screen is a pleasant surprise. A `Column`, a few `Text` and `Button` calls, a `remember { mutableStateOf(...) }`, and the UI updates itself. No XML, no adapters, no view holders. The trouble starts later, when the app grows. A text field drops characters while you type. A list recomposes every row when one item changes. An effect runs twice, or never stops. A layout that looked fine on a phone wastes half the screen on a tablet. The theme looks right in light mode and unreadable in dark mode. An animation stutters, a screen reader announces nonsense, and nobody can say which part of the code is to blame.

Almost all of these problems have the same cause: Compose is easy to write and harder to reason about. Its rules for composition, recomposition, state, the layout phases, and the modifier chain are consistent, but most tutorials never spell them out. Once you know them, the surprises go away. You can predict when a composable runs again and when it is skipped, where state should live and how long it survives, what size a layout will choose, which phase a state read invalidates, and why one animation is smooth while another is not. Then you can build the custom layouts, modifiers, themes, and interactions that make an app feel finished.

This book is about that second stage. We assume you have already built a small Compose screen, and we go under the surface: how Compose works, how layout is negotiated, how to structure state for real screens, how to theme with Material 3 and navigate with type-safe routes, how to animate and handle gestures, and how to make the result accessible, fast, tested, and ready for phones, tablets, and foldables.

Who This Book Is For

- **Android developers who know the basics of Compose** and want to understand why it behaves the way it does, not just which modifier to add next.
- **View-system developers moving to Compose** who need the mental model, the interop story, and production patterns rather than another beginner tour.
- **Developers maintaining a growing Compose codebase** who are fighting recomposition problems, fragile layouts, state bugs, or inconsistent components.
- **Developers who care about quality** and want adaptive layouts, real accessibility, measurable performance, and tests that catch UI regressions.

You should be comfortable with Kotlin (classes, lambdas, null safety, basic coroutines and Flow) and with Android Studio. You do not need to know the View system, although we compare with it where that helps.

What You Will Learn

- How Compose turns your functions into a running interface: the compiler plugin, composition, recomposition, skipping, the snapshot system, and the three phases of a frame.
- State in depth: `remember`, derived state, hoisting, side effects and their lifecycles, ViewModels with unidirectional data flow, CompositionLocals, and saving state across configuration changes and process death.
- Layout in depth: constraints, intrinsics, lazy lists and grids, custom `Layout` and `SubcomposeLayout`, custom modifiers built on `Modifier.Node`, adaptive layouts with window size classes, the navigation suite, and list-detail panes, plus custom drawing with Canvas.
- Material 3 theming with brand and dynamic color, a small design system with custom tokens, everyday components, and edge-to-edge insets.
- Navigation Compose with type-safe routes, nested graphs, top-level destinations, deep links, and returning results.
- Animation and interaction: value animations, visibility and content changes, transitions, `Animatable`, infinite and shared element transitions, gestures from clicks to custom pointer input.
- Production quality: accessibility semantics and contrast, stability and performance, previews, Compose UI tests and screenshot tests, and interoperability with existing Views.

How This Book Is Organized

Part I — How Compose Works (Chapters 1–5) builds the mental model: composition, recomposition, state, modifiers, and side effects. **Part II — Layout and Drawing** (Chapters 6–10) explains measurement and goes all the way to custom layouts, custom modifiers, adaptive layouts, and Canvas drawing. **Part III — State and Architecture for UI** (Chapters 11–15) covers state holders and ViewModels, forms and text input, CompositionLocals, asynchronous data, and saving state. **Part IV — Material 3 and Navigation** (Chapters 16–20) covers theming, a design system, components, and navigation architecture. **Part V — Motion and Interaction** (Chapters 21–24) covers animation and gestures and ends by building the live recording screen. **Part VI — Production Quality** (Chapters 25–28) covers accessibility, performance, previews and testing, and View interop, finishing with a review of the whole app. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with **Try It Yourself** exercises and a **Key Takeaways** box for quick review.

The Running Example: Waypoint

Throughout the book we build the user interface of **Waypoint**, a hiking trip planner and tracker made by a small fictional product studio, **Northbeam Labs**. Hikers browse and search eight sample trails, open a trail to see its distance, climb, and estimated time, plan a trip with a date, a party size, and a checklist, record a hike with a live timer and an elevation profile, keep a private journal, and choose

their units and theme in Settings. Waypoint is small enough to understand completely and rich enough to need everything in this book: custom layouts and modifiers, a list-detail layout for tablets, a form with validation, Material 3 theming with custom tokens, type-safe navigation with deep links, animations and gestures, accessibility for custom controls, and tests. The sample data is fixed and small, so you can check every number in the book by hand.

You start from a small **Waypoint starter project** with the data models, sample data, in-memory repositories, a theme, and a first trail list. Waypoint is also the running example of the other Android books from thenexbooks, but this book is self-contained: you never need another book to follow it.

Versions and Conventions

At the time of writing, the code uses **Kotlin 2.4** with the Compose compiler Gradle plugin, the **Compose BOM 2026.09.00** (Compose UI 1.12 and Material 3 1.4), **Navigation Compose 2.10** with type-safe routes, Android Gradle Plugin 9.4, and JDK 21, with a minimum SDK of 26 and a target SDK of 36. Compose evolves quickly; where a detail may differ in your version, a Note points it out. The examples were compiled and their tests run on the JVM with Robolectric and Roborazzi. Behavior that needs a device, such as TalkBack, benchmarks, or a real foldable, is described as "something like" what you will see, because exact output varies between devices and releases.

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

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

> **WARNING**
> A common mistake or something that can cause bugs, crashes, or poor performance.

Code appears in `monospace` with a caption naming the file, such as `app/src/main/kotlin/com/northbeam/waypoint/feature/trails/TrailListScreen.kt`, sometimes shortened to `.../feature/trails/TrailListScreen.kt`. Import lists are omitted when they add nothing. Screens are described in words and with simple text sketches.

Contents

Preface

PART I — HOW COMPOSE WORKS

1 Thinking in Compose: The Mental Model and the Waypoint Starter

2 Composition and Recomposition: What the Compiler Generates and When Code Runs Again

3 State in Depth: remember, Snapshots, State Hoisting, and Derived State

4 Modifiers in Depth: Order, the Modifier Chain, and the Three Phases

5 Side Effects and Lifecycles: LaunchedEffect, DisposableEffect, and Friends

PART II — LAYOUT AND DRAWING

6 The Layout Model: Constraints, Measurement, and Intrinsics

7 Lists and Grids: Lazy Layouts, Keys, Sticky Headers, and Scroll State

8 Custom Layouts and Custom Modifiers: Layout, SubcomposeLayout, and Modifier.Node

9 Adaptive Layouts: Window Size Classes, Navigation Suite, List-Detail, and Foldables

10 Drawing and Graphics: Canvas, DrawScope, Paths, and graphicsLayer

PART III — STATE AND ARCHITECTURE FOR UI

11 State Holders and ViewModels: UI State, Events, and Unidirectional Data Flow

12 Forms and Text Input: The Trip Planner with TextFieldState, Pickers, and Validation

13 CompositionLocals and Passing Dependencies Through the Tree

14 Async Data in the UI: Loading, Errors, Pull-to-Refresh, and Snackbars

15 Saving and Restoring State: Configuration Changes, Process Death, and SavedStateHandle

PART IV — MATERIAL 3 AND NAVIGATION

16 Material 3 Theming: Color Schemes, Dynamic Color, Typography, and Shapes

17 Building a Design System: Custom Tokens, Dark Theme, and Reusable Components

18 Material 3 Components in Practice: Scaffold, App Bars, Sheets, Dialogs, and Insets

19 Navigation Compose with Type-Safe Routes

20 Navigation Architecture: Nested Graphs, Top-Level Destinations, Deep Links, and Results

PART V — MOTION AND INTERACTION

21 Animation Fundamentals: animate*AsState, AnimatedVisibility, and AnimatedContent

22 Advanced Animation: Transitions, Animatable, Infinite Motion, and Shared Elements

23 Gestures and Touch: Clicks, Drags, Swipe-to-Dismiss, and pointerInput

24 Building the Recording Screen: Live Data, Timers, and Meaningful Motion

PART VI — PRODUCTION QUALITY

25 Accessibility in Depth: Semantics, TalkBack, Touch Targets, and Contrast

26 Performance: Stability, Strong Skipping, Deferred Reads, and Measuring

27 Previews and Testing: Multipreview, Compose UI Tests, and Screenshot Tests

28 Views Interop and Shipping Waypoint: AndroidView, ComposeView, Migration, and a Final Review

APPENDICES

A Jetpack Compose Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

How Compose Works

The mental model behind every Compose screen: what the compiler does to a composable function, when code runs again and when it is skipped, how the snapshot system tracks state, how modifiers form a chain that acts in the composition, layout, and drawing phases, and how to run side effects safely.

1 · Thinking in Compose

2 · Composition and Recomposition

3 · State in Depth

4 · Modifiers in Depth

5 · Side Effects and Lifecycles

CHAPTER 1 · PART I — HOW COMPOSE WORKS

Thinking in Compose: The Mental Model and the Waypoint Starter

Before we tune recomposition, write custom layouts, or animate shared elements, we need a precise picture of what a composable function is and what Compose does with it. This chapter builds that mental model, sets up the toolchain, and walks through the Waypoint starter project that every later chapter extends.

Meet Northbeam Labs and Waypoint

Throughout this book we build the user interface of **Waypoint**, a hiking trip planner and tracker made by **Northbeam Labs**, a small (and fictional) product studio. Waypoint lets hikers browse trails, read a trail's distance, climb, and estimated time, plan a trip with a date, a party size, and a packing checklist, record a hike while they walk, and keep a private journal afterward. The team is four people. Lena Ortiz leads Android development, Sam Okafor owns the backend, Priya Natarajan designs the product, and Theo Brandt runs QA. You will see them in short scenarios: Priya hands over a mockup, Theo files an accessibility bug, Lena decides how a screen should be split between a ViewModel and a composable.

The app has every kind of UI problem a real product has: a searchable scrolling list, a dense statistics layout, a validated form, live data that changes every second, app-wide theming, and side-by-side panes on tablets and foldables. Each maps to a part of Compose.

This book focuses on the UI layer. Waypoint's data comes from small in-memory repositories seeded with fixed sample data, so you never need a server, a database, or a dependency-injection framework to follow along. Other books in the series, such as "Android App Architecture", replace those pieces with production versions without changing a single composable.

From Imperative to Declarative UI

If you have built Android screens with XML layouts and the View system, you have written *imperative* UI code. You inflate a tree of views once, keep references to the ones you care about, and then mutate them as things change: set this text, hide that spinner, change the icon on this button. The view tree is a long-lived object graph, and your job is to keep it in sync with your data by hand.

Here is a tiny example in the old style. A trail row shows a heart icon and a favorite count; tapping the heart must update both.

LegacyTrailRow.kt (View system, for contrast only)

```
// Imperative: we own the widgets and must remember every mutation.
favoriteButton.setOnClickListener {
    isFavorite = !isFavorite
    favoriteCount += if (isFavorite) 1 else -1
    favoriteButton.setImageResource(
        if (isFavorite) R.drawable.ic_favorite else R.drawable.ic_favorite_border,
    )
    favoriteButton.contentDescription =
        if (isFavorite) "Remove from favorites" else "Add to favorites"
    countView.text = "$favoriteCount favorites"   // forget this line and the UI lies
}
```

Nothing in this code is hard, but every line is an opportunity for the screen and the data to disagree. If a second code path changes `isFavorite` (a sync from the server, an undo action), it must repeat the same three mutations. Bugs in View-based apps are very often exactly this: a widget that was not updated on one of the paths that changed its data.

Compose turns the problem around. You do not mutate widgets. You write a function that describes what the UI should look like *for a given state*, and Compose takes care of making the screen match that description whenever the state changes.

.../ui/components/FavoriteSummary.kt (illustration)

```
@Composable
fun FavoriteSummary(
    isFavorite: Boolean,
    favoriteCount: Int,
    onToggle: () -> Unit,
    modifier: Modifier = Modifier,
) {
    Row(modifier, verticalAlignment = Alignment.CenterVertically) {
        IconButton(onClick = onToggle) {
            Icon(
                imageVector = if (isFavorite) Icons.Filled.Favorite
                else Icons.Filled.FavoriteBorder,
                contentDescription =
                    if (isFavorite) "Remove from favorites" else "Add to favorites",
            )
        }
        Text("$favoriteCount favorites")
    }
}
```

There is no "set" anywhere. The icon, the description, and the text are all expressions of the two inputs. When `isFavorite` changes, whoever owns that value changes it in one place, and Compose calls the function again with the new value. It is impossible for the icon to say "favorite" while the count says otherwise, because both are computed from the same data in the same call.

UI = f(state)

The shorthand for this idea is **UI = f(state)**: the user interface is a function of the application state. It is a slogan, but it carries three precise consequences that the rest of the book depends on.

1. **State is the only input.** If something on screen can change, there must be a piece of state that describes it. If you cannot point to that state, the UI will eventually be wrong.
2. **The function can run again at any time.** Compose may call your composable once, or a hundred times, and the result must be the same for the same state. That rules out hidden work such as writing to a database from inside the function body.
3. **Changing state is how you change the UI.** Events (a click, a keystroke, a network response) update state; Compose notices and re-runs the functions that read that state. We call that re-run *recomposition*, and Chapter 2 is devoted to it.

The loop that results is the heart of every Compose app: state flows *down* into composables as parameters, and events flow *up* out of them as lambda calls. Whoever owns the state handles the event, produces new state, and the UI follows. Chapter 11 formalizes this as unidirectional data flow, but you will see it in every listing from this chapter on.



What a Composable Really Is

A composable is a Kotlin function annotated with `@Composable`. The annotation looks small, but it changes the function's type: a composable can only be called from another composable (or from a composable lambda), much like a `suspend` function can only be called from a coroutine. The Compose compiler plugin rewrites every composable so that it can take part in composition, and Chapter 2 opens that box. For now, these are the properties you should hold in mind whenever you write one.

It emits UI instead of returning it

Composables that draw something return `Unit`. `TrailCard` does not return a card object; calling it *emits* a card into the tree that Compose is building. Compose remembers what was emitted at each position in the code, which is how it can later compare the new description with the old one and update only what changed. Composables that compute a value (for example `remember` or `MaterialTheme.colorScheme`) do return something, and by convention their names start with a lowercase letter. Composables that emit UI are named like types, with a capital letter: `Text`, `TrailCard`, `TrailListContent`.

It is idempotent

Called twice with the same arguments, a composable must produce the same UI. That is what allows Compose to skip calls whose inputs have not changed, and to call a function again whenever it needs to without worrying about history.

It is free of side effects

A side effect is any change visible outside the function: writing a file, logging analytics, mutating a global variable, starting a network request, navigating. A composable body should not do any of those things directly, because you do not control when or how often it runs. When you genuinely need a side effect (and you will, for timers, listeners, and snackbars), Compose provides controlled APIs such as `LaunchedEffect` and `DisposableEffect`, covered in Chapter 5.

It is fast

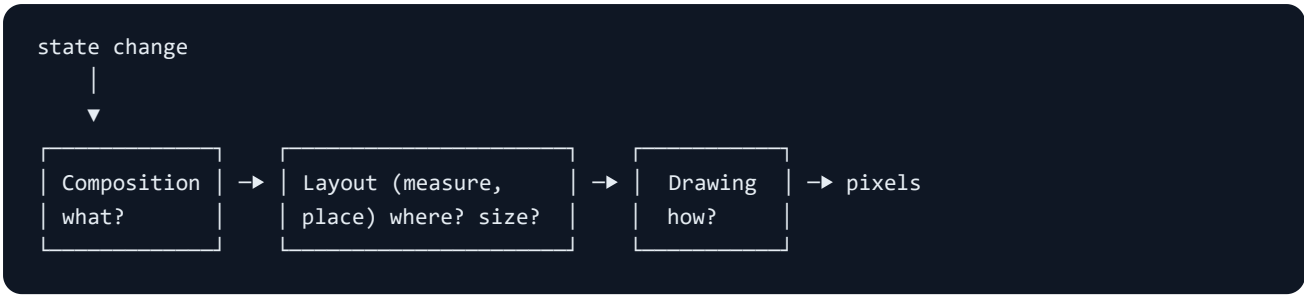
Composition can happen as often as every frame during an animation. A composable body should do little more than read its parameters and call other composables. Sorting a list of trails, parsing a date, or formatting thousands of strings belongs elsewhere: in a ViewModel, in a `remember` block keyed on its inputs, or in a plain function called once when data arrives. Waypoint's trail list is sorted and filtered in the `TrailListViewModel`, not in `TrailListContent`, for exactly this reason.

> **WARNING**
>
> The most common beginner bug in Compose is code that "works" in a preview but misbehaves in the app because it does work in the function body: a counter incremented during composition, a log call that fires dozens of times, a request launched every time the screen recomposes. If a line of code in a composable changes something outside the function, stop and ask which effect API it belongs in.

The Three Phases at a Glance

Turning state into pixels happens in three phases, every frame that something changed. Knowing which phase a piece of code runs in is the single most useful performance and debugging skill in Compose, so we name the phases now and come back to them repeatedly.

1. **Composition** answers "*what* UI should exist?" Compose runs your composable functions and builds (or updates) a tree of layout nodes: a `Scaffold` containing a `Column` containing a text field, a row of chips, and a lazy list of cards.
2. **Layout** answers "*where* does each node go, and how big is it?" Each node is measured with constraints from its parent, reports a size, and then places its children. Measurement and placement are two steps of the same phase.
3. **Drawing** answers "*how* does it look?" Each node draws itself (text, backgrounds, borders, icons) into a canvas, in the positions computed during layout.



The important detail is that Compose tracks *which phase read which state*. If a piece of state is only read while drawing (say, the color of a progress bar inside a `drawBehind` lambda), changing it only re-runs drawing; composition and layout are skipped for that frame. If it is read during layout (for example inside an `offset { }` lambda), only layout and drawing run. Chapter 4 shows how modifiers let you choose the phase in which a value is read, and Chapter 26 turns that into a performance technique called deferred reads.

The Toolchain at the Time of Writing

Compose moves quickly, so we pin every version in one place and tell you exactly what the sample code was built with. At the time of writing, the Waypoint project uses the versions below. Newer versions will almost certainly exist by the time you read this; the concepts in this book do not depend on them, and where an API is new or still changing we say so.

Component	Version used	Notes
Kotlin	2.4.20	Compose compiler plugin has the same version
Compose BOM	2026.09.00	UI, foundation, runtime, animation 1.12.1
Material 3	1.4.0	material3 adaptive 1.3.0
Navigation Compose	2.10.2	type-safe `@Serializable` routes
Lifecycle / Activity Compose	2.11.0 / 1.13.0	ViewModel, lifecycle-aware collection
Android Gradle Plugin / Gradle	9.4.1 / 9.8.0	built-in Kotlin support
JDK	21	bytecode target 17
SDK levels	compile 37, target 36, min 26	minSdk 26 has `java.time`

The Compose compiler is a Kotlin compiler plugin

Compose is not an annotation processor and it does not generate source files. The `@Composable` transformation is done by a **Kotlin compiler plugin** with the id `org.jetbrains.kotlin.plugin.compose`. Since Kotlin 2.0 the plugin ships with Kotlin itself and has

exactly the same version number, so there is no longer a separate compatibility table to consult: when you upgrade Kotlin, you upgrade the Compose compiler. In the project, the plugin is declared in the version catalog and applied to the app module.

gradle/libs.versions.toml (excerpt)

```
[versions]
agp = "9.4.1"
kotlin = "2.4.20"
composeBom = "2026.09.00"   # ui/foundation/runtime/animation 1.12.1, material3 1.4.0
navigationCompose = "2.10.2"
lifecycle = "2.11.0"

[libraries]
compose-bom = { module = "androidx.compose:compose-bom", version.ref = "composeBom" }
compose-ui = { module = "androidx.compose.ui:ui" }
compose-foundation = { module = "androidx.compose.foundation:foundation" }
compose-material3 = { module = "androidx.compose.material3:material3" }
compose-ui-tooling-preview = { module = "androidx.compose.ui:ui-tooling-preview" }

[plugins]
android-application = { id = "com.android.application", version.ref = "agp" }
kotlin-compose = { id = "org.jetbrains.kotlin.plugin.compose", version.ref = "kotlin" }
```

The BOM keeps Compose libraries in step

Compose is split into many artifacts (runtime, UI, foundation, animation, Material 3, test libraries). They are released together and are meant to be used at matching versions. The **Bill of Materials** (BOM) is a special artifact that contains no code, only a list of versions. You import it as a `platform`, and then declare each Compose library *without* a version, as you can see in the catalog above.

app/build.gradle.kts (excerpt)

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.compose)
    alias(libs.plugins.kotlin.serialization)   // @Serializable navigation routes
    alias(libs.plugins.roborazzi)
}

android {
    buildFeatures { compose = true }
    // compileSdk, minSdk, targetSdk come from the version catalog
}

dependencies {
    implementation(platform(libs.compose.bom))
    implementation(libs.compose.ui)
    implementation(libs.compose.foundation)
    implementation(libs.compose.material3)
    implementation(libs.compose.ui.tooling.preview)
    debugImplementation(libs.compose.ui.tooling)
```

```
    testImplementation(platform(libs.compose.bom))
    testImplementation(libs.compose.ui.test.junit4)
}
```

Notice that the BOM is imported twice: once for `implementation` and once for `testImplementation`. Each configuration resolves its own dependencies, so a test library such as `ui-test-junit4` needs the platform too. Forgetting the second line is a classic cause of a confusing "could not find androidx.compose.ui:ui-test-junit4:" error.

> **TIP**
>
> When a Compose library you need is not covered by the BOM (or you want a newer version of one artifact for a bug fix), you can still give it an explicit version. An explicit version wins over the BOM for that artifact only. Use this sparingly, and remove the override when the next BOM catches up.

A Tour of the Waypoint Starter

Every chapter begins from the **Waypoint starter**, a small but complete app that already builds, runs, and passes its tests. Opening it in Android Studio and reading it once, end to end, is the best preparation for the rest of the book. Here is the layout.

```
waypoint/
├── settings.gradle.kts          :app, :core:model, :core:data
├── gradle/libs.versions.toml    every version in one place
├── core/model/   (Kotlin/JVM)   Trail, Trip, SampleData, WaypointFormat, TrailMath, ...
├── core/data/    (Kotlin/JVM)   repository interfaces + InMemory* implementations
└── app/          (Android)
    ├── WaypointApplication.kt   AppContainer, DefaultAppContainer
    ├── MainActivity.kt          enableEdgeToEdge, WaypointTheme, WaypointApp()
    ├── navigation/              TrailListRoute, TrailDetailRoute, WaypointNavHost
    ├── ui/theme/                Color.kt, Theme.kt
    ├── ui/components/           TrailCard, DifficultyChip, StatTile
    └── feature/trails/          TrailList*, TrailDetail*
```

core:model: the data the UI shows

The `core:model` module is plain Kotlin with no Android dependency. It holds the models shared by every book in the series (`Trail`, `Trip`, `TrackPoint`, `JournalEntry`, `User`) plus a few book-specific types such as `UserSettings`, `TrailFilter`, and `TripPlanDraft`.

`core/model/src/main/kotlin/com/northbeam/waypoint/core/model/Models.kt (excerpt)`

```
data class Trail(
    val id: String,
    val name: String,
    val region: String,
    val distanceKm: Double,
```

```
    val elevationGainM: Int,
    val difficulty: Difficulty,
    val rating: Double,
    val photoUrl: String?,
)

enum class Difficulty { EASY, MODERATE, HARD }
```

The same module contains `SampleData`, the fixed data set used by previews, tests, and every number quoted in this book. It has eight trails, from `tr-001` Cedar Loop (Pine Ridge, 4.2 km, 120 m of climb, Easy, rated 4.6) to `tr-008` Brookside Way (Mirror Lake, 8.0 km, 210 m, Moderate, 4.4), five trips, three journal entries, and seven track points for the hike in progress. It also fixes "now": `SampleData.today` is Friday, June 12, 2026, and `SampleData.referenceNow` is 09:00 UTC that morning. Because time is fixed, a screenshot taken today and one taken next year are identical.

Two small objects hold the display rules. `WaypointFormat` turns numbers into strings (`distanceKm(4.2)` is `"4.2 km"`, `elevation(640)` is `"+640 m"`, `duration(182)` is `"3 h 2 min"`), and `TrailMath` computes estimates: the estimated time of a trail is 12 minutes per kilometer plus one minute per 10 meters of climb, rounded. For Falcon Crest Trail that is 9.8 × 12 + 640 / 10 = 181.6, rounded to 182 minutes, displayed on its card as "~3 h 2 min".

core:data: repositories behind interfaces

The UI never touches `SampleData` directly at runtime. It talks to repository interfaces such as `TrailRepository`, which exposes a `Flow<List<Trail>>` of trails, a flow of favorite ids, and suspend functions to toggle a favorite or refresh. The starter ships in-memory implementations seeded with the sample data. `InMemoryTrailRepository` can simulate network latency and a failing refresh, which Chapter 14 uses to build loading and error states without a real server.

The app container

The app module wires repositories together with a deliberately tiny manual container. There is no Hilt here; the goal is to keep every listing about Compose.

app/src/main/kotlin/com/northbeam/waypoint/WaypointApplication.kt (excerpt)

```
interface AppContainer {
    val trailRepository: TrailRepository
    val tripRepository: TripRepository
    val journalRepository: JournalRepository
    val settingsRepository: SettingsRepository
    val userRepository: UserRepository
    val clock: Clock
}

class DefaultAppContainer(
    override val clock: Clock = Clock.systemDefaultZone(),
) : AppContainer {
    override val trailRepository: TrailRepository =
        InMemoryTrailRepository(latencyMs = 300)
```

```
    override val tripRepository: TripRepository = InMemoryTripRepository()
    // journal, settings, and user repositories omitted
}

class WaypointApplication : Application() {
    lateinit var container: AppContainer
        private set

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

The `clock` property matters more than it looks. Any ViewModel that needs "today" reads `LocalDate.now(clock)`, so tests and previews can pass `Clock.fixed(SampleData.referenceNow, ZoneOffset.UTC)` and get repeatable results.

MainActivity: the entry point

`MainActivity` is short. It enables edge-to-edge drawing, reads the user's settings, chooses light or dark colors, and hands control to the root composable.

app/src/main/kotlin/com/northbeam/waypoint/MainActivity.kt

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        val container = (application as WaypointApplication).container
        setContent {
            val settings by container.settingsRepository.settings
                .collectAsStateWithLifecycle()
            val darkTheme = when (settings.themeMode) {
                ThemeMode.SYSTEM -> isSystemInDarkTheme()
                ThemeMode.LIGHT -> false
                ThemeMode.DARK -> true
            }
            WaypointTheme(
                darkTheme = darkTheme,
                dynamicColor = settings.useDynamicColor,
            ) {
                WaypointApp()
            }
        }
    }
}

@Composable
fun WaypointApp() {
    val navController = rememberNavController()
    WaypointNavHost(navController = navController)
}
```

`setContent` is the bridge between the Activity world and the Compose world: everything inside the lambda is composition. Notice that this is already UI = f(state) at the scale of the whole app. The theme is not "switched" by imperative code; it is computed from `settings`, and when the settings flow emits a new value, the theme follows. Chapter 16 builds the Settings screen that changes those values.

The starter's navigation is minimal: `WaypointNavHost` has two destinations, `TrailListRoute` and `TrailDetailRoute(trailId)`, defined as `@Serializable` types. Chapters 19 and 20 grow this into the full app graph.

The Screen and Content split

The most important pattern in the starter is the way each screen is split into two composables. Look at the trail list.

.../feature/trails/TrailListScreen.kt (excerpt)

```
/** Stateful entry point: owns the ViewModel and forwards plain values. */
@Composable
fun TrailListScreen(
    onTrailClick: (String) -> Unit,
    modifier: Modifier = Modifier,
    viewModel: TrailListViewModel = viewModel(factory = TrailListViewModel.Factory),
) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    TrailListContent(
        uiState = uiState,
        onQueryChange = viewModel::onQueryChange,
        onDifficultyToggle = viewModel::onDifficultyToggle,
        onTrailClick = onTrailClick,
        onFavoriteToggle = viewModel::onFavoriteToggle,
        modifier = modifier,
    )
}

/** Stateless trail list: easy to preview, test, and screenshot. */
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun TrailListContent(
    uiState: TrailListUiState,
    onQueryChange: (String) -> Unit,
    onDifficultyToggle: (Difficulty) -> Unit,
    onTrailClick: (String) -> Unit,
    onFavoriteToggle: (String) -> Unit,
    modifier: Modifier = Modifier,
) {
    Scaffold(
        modifier = modifier,
        topBar = { TopAppBar(title = { Text("Trails") }) },
    ) { innerPadding ->
        // search field, difficulty chips, then the list of TrailCards
    }
}
```

`TrailListScreen` is the **stateful** entry point. It obtains the ViewModel, collects its `StateFlow<TrailListUiState>` in a lifecycle-aware way, and passes plain values and lambdas down. `TrailListContent` is **stateless**: everything it shows comes from `uiState`, and everything the user does goes out through an `onXxx` lambda. It has no idea that a ViewModel exists.

The payoff is large. A stateless content composable can be previewed with any state you like (loading, empty, eight trails, an error), tested on the JVM by passing a fake state and recording lambda calls, and reused on a tablet layout that hosts it next to a detail pane. The stateful wrapper is so thin that it barely needs testing. We will use the same split for every screen in the book: `TrailDetailScreen` and `TrailDetailContent`, `TripPlannerScreen` and `TripPlannerContent`, and so on.

What does the starter's trail list look like? A top app bar titled "Trails"; an outlined text field labeled "Search trails or regions" with a search icon; a horizontal row of three filter chips (Easy, Moderate, Hard); and below them a vertical list of elevated cards with 16 dp of side padding and 12 dp between cards. Each `TrailCard` has a 48 dp rounded square showing the trail's first letter, the name in a bold title style over the region, and a heart button on the right. A second row holds a small difficulty badge, a star with the rating, the distance, the climb, and the estimated time: for Falcon Crest Trail, "Hard", "4.8", "9.8 km", "+640 m", "~3 h 2 min". The cards are ordered by rating, so Falcon Crest Trail (4.8) is first and Meadow Run (4.0) is last.

Tapping a card navigates to `TrailDetailContent`: a top bar with a back arrow and the trail name, the region, the difficulty badge, a row of three statistic tiles (Distance, Elevation, Est. time), and a full-width "Plan a trip" button. The starter is honest about its flaws, and one of them is visible here: on a narrow phone, the "Est. time" tile for Granite Steps wraps "2 h 11 min" onto two lines, so the three tiles no longer line up. Chapter 6 explains why, and Chapter 8 fixes it properly with a custom layout.

Tests that already pass

The starter includes tests in all three modules, and they run on your computer's JVM without an emulator:

- `WaypointModelTest` (core:model) pins the sample data and pre-computed numbers.
- `InMemoryRepositoriesTest` (core:data) checks the fake repositories.
- `TrailListViewModelTest` (app) checks filtering and the refresh error with Turbine.
- `TrailListContentTest` (app) is a Compose UI test under Robolectric: it clicks a card, toggles a chip, types a query, and asserts which lambdas were called.
- `TrailScreensScreenshotTest` (app) compares the list and detail screens to golden PNGs with Roborazzi.
- `MainActivityTest` (app) launches the real Activity as a smoke test.