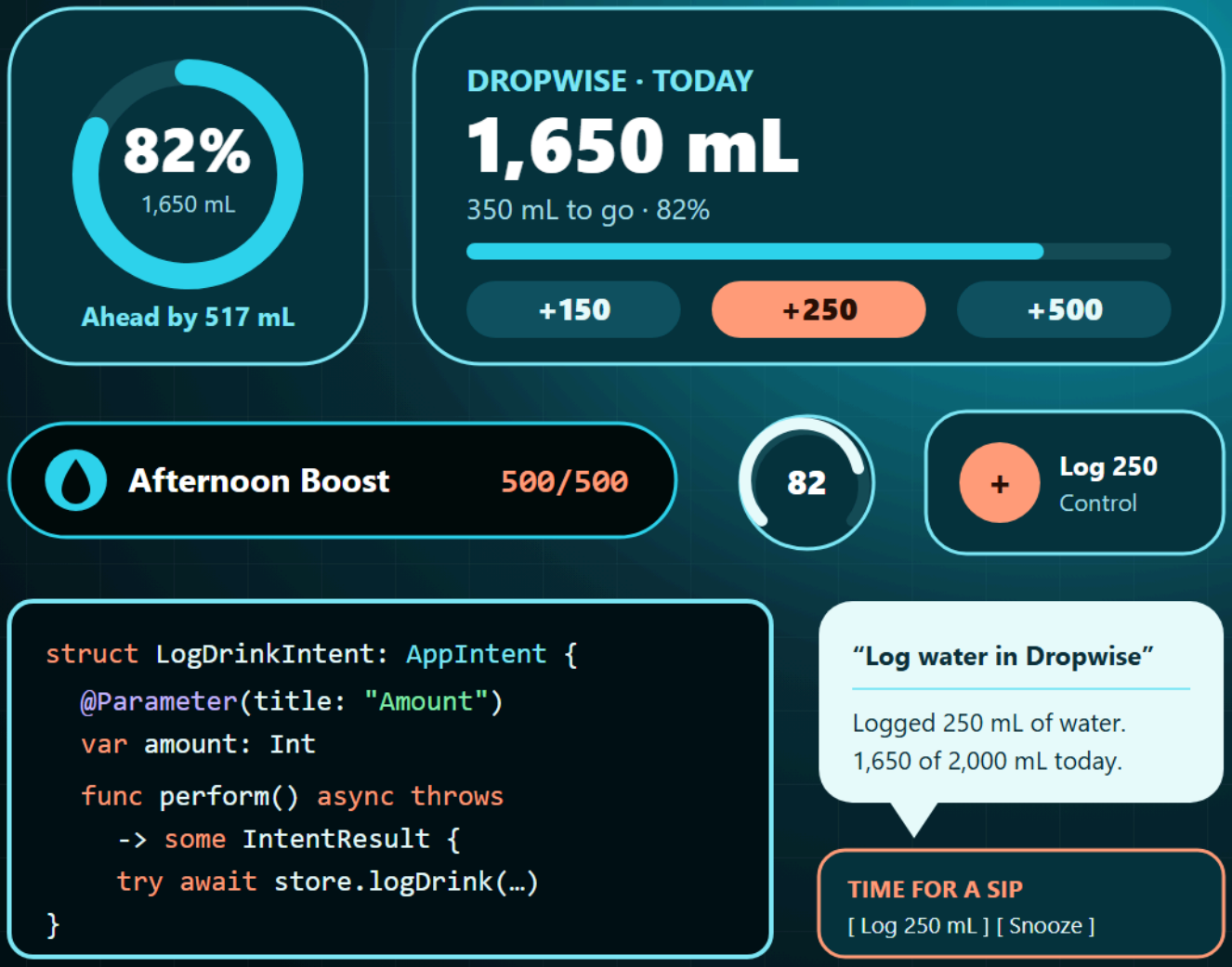


EXTEND YOUR iOS APP · SWIFT 6

Widgets, Live Activities, and App Intents

Extending your iOS app beyond the screen — one surface at a time

WidgetKit · ActivityKit · Dynamic Island · Siri · Shortcuts
Spotlight · Controls · Focus · Notifications · Share

WIDGETS · LIVE ACTIVITIES · INTENTS · CONTROLS

THENEXBOOKS

Widgets, Live Activities, and App Intents

Extending Your iOS App Beyond the Screen: WidgetKit, ActivityKit, Shortcuts, Siri, Controls, and Notifications

From one app to every surface of the system — built step by step with Swift 6

First Edition

THENEXBOOKS

Widgets, Live Activities, and App Intents
Extending Your iOS App Beyond the Screen: WidgetKit, ActivityKit, Shortcuts, Siri, Controls, and Notifications

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. Apple's platforms, SDKs, and tools change frequently; APIs, system behaviors, refresh budgets, limits, and review rules can differ between releases and may change after publication. Always consult Apple's current documentation for the versions you use. System behaviors described in this book, such as when widgets refresh or when background tasks run, are decided by the operating system and are described as typical behavior, not guarantees.

Apple, iPhone, iPad, Apple Watch, Mac, macOS, watchOS, Siri, Spotlight, Xcode, Swift, SwiftUI, Dynamic Island, StandBy, Apple Intelligence, CarPlay, App Store, and TestFlight are trademarks of Apple Inc., registered in the U.S. and other countries and regions. IOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license. 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 Apple Inc. All widget, Live Activity, and interface mockups in this book and on its cover are original illustrations.

Dropwise, the hydration and habit tracker used as the running example in this book, is fictional, as are its data, server, and users. Domain names use the reserved `.example` domain. Dropwise's daily goals are user-chosen numbers for demonstration only; nothing in this book is medical or health advice.

Preface

Most of the time people spend with your app happens outside it. They glance at a widget on the Home Screen, check a Live Activity on the Lock Screen while their phone is still in their hand, ask Siri to do something without unlocking, tap a control in Control Center, answer a notification with one button, or share a link from Safari. Each of those moments takes a second or two, and in each of them your app is not running in the usual sense at all. The system is showing something your code prepared earlier, or it is running a small piece of your code in a separate process, with tight limits on time and memory.

That is why extending an app into the system feels so different from building screens inside it. A widget is not a tiny app; it is a timeline of snapshots. A Live Activity is not a view you update whenever you like; it is content the system displays within a budget, often updated from a server. An App Intent is not a button handler; it is a description of an action that Siri, Shortcuts, Spotlight, widgets, and controls can all invoke. Once you understand the model behind each surface, the APIs become straightforward. Without that model, they are a source of confusing bugs: widgets that never update, data that does not appear in an extension, Live Activities that go stale, and intents that work in the app but not from the Lock Screen.

This book teaches those models and the code that follows from them. We take one existing SwiftUI app and extend it, surface by surface, until it lives everywhere iOS allows: widgets in every family, an interactive widget, a configurable widget, a Live Activity with Dynamic Island layouts and push updates, App Intents with entities and App Shortcuts for Siri, Spotlight indexing, Control Center controls, a Focus filter, actionable and time-sensitive notifications with service and content extensions, a Share extension, and background refresh that keeps everything consistent.

Who This Book Is For

- **iOS developers who already build SwiftUI apps** and want them to appear on the Home Screen, Lock Screen, Dynamic Island, Control Center, Siri, and Spotlight.
- **Developers maintaining an older widget or SiriKit integration** who need to move to WidgetKit's current APIs, interactive widgets, and App Intents.
- **Teams planning Live Activities with server-driven updates**, who need to understand both the app side and the APNs side.
- **Experienced developers from other platforms** who know Swift and SwiftUI basics and want a structured guide to Apple's extension model.

We assume you are comfortable with Swift, SwiftUI views and state, and async/await. If you are new to SwiftUI, learn the basics first; this book does not teach them again.

What You Will Learn

- How app extensions work: separate processes, App Groups, a shared store, Swift 6 concurrency across targets, and deep links back into the app.
- How to build widgets with WidgetKit: timeline providers, every family, Lock Screen and StandBy layouts, tinted rendering, refresh strategies within the system's budget, relevance, interactive buttons and toggles, and configurable widgets.
- How to build Live Activities with ActivityKit: attributes and content state, Lock Screen and Dynamic Island layouts, local and push updates, push-to-start, broadcast updates, buttons, and the Apple Watch Smart Stack.
- How to design App Intents: parameters, enums, entities, queries, dialogs, results, App Shortcuts for Siri, Spotlight indexing, and the newer Apple Intelligence integrations.
- How to add controls for Control Center, the Lock Screen, and the Action button, plus Focus filters.
- How to build rich notifications: categories, actions, interruption levels, time-sensitive alerts, service extensions, and content extensions — and a Share extension.
- How to ship all of it: background refresh, keeping every surface in sync, testing and debugging extensions, privacy manifests, performance limits, and accessibility.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains the extension model and sets up the project, the App Group, the shared store, concurrency rules, and deep links. **Part II — WidgetKit** (Chapters 6–10) builds static, interactive, and configurable widgets for every family. **Part III — Live Activities** (Chapters 11–14) covers ActivityKit from the first local activity to push-to-start and broadcast updates. **Part IV — App Intents and Siri** (Chapters 15–19) covers intents, parameters, entities, App Shortcuts, Siri, Spotlight, and Apple Intelligence. **Part V — Controls, Notifications, and Sharing** (Chapters 20–24) covers controls, Focus filters, notifications and their extensions, and the Share extension. **Part VI — Production** (Chapters 25–28) covers background work, testing and debugging, privacy and performance, and a complete case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

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

The Running Example: Dropwise

Throughout the book we extend **Dropwise**, a fictional hydration and habit tracker. Its users log drinks against a daily goal they choose (2,000 mL in our sample data) and tick off a few daily habits such as a morning stretch, vitamins, and reading ten pages. At the start of the book Dropwise is an ordinary SwiftUI app. By the end, its progress ring sits on the Home Screen and Lock Screen, a medium widget logs a glass with one tap, an "Afternoon Boost" challenge runs as a Live Activity in the Dynamic Island, Siri answers "How much have I had in Dropwise?", a control in Control Center logs

250 mL, a Focus filter pauses reminders, notifications offer a "Log 250 mL" button, and a Share extension saves habit ideas from Safari. One sample day — Tuesday, March 10, 2026 — runs through the whole book, so every number on every surface can be checked by hand.

Versions and Conventions

The code uses the **Swift 6 language mode** with strict concurrency checking, **Xcode 26-era tooling**, and a **deployment target of iOS 18**. Features that require iOS 26 are marked with availability checks and described in Notes; because Apple's platforms change every year, treat those sections as an introduction and check the current documentation for details. You need a Mac with a recent Xcode; a physical iPhone is strongly recommended, because some surfaces, such as the Dynamic Island on supported models, Live Activity push updates, and the Action button, are best tested on a device.

The system decides when widgets refresh, when background tasks run, and how many updates a Live Activity may receive. Apple does not publish exact budgets, so this book describes typical behavior and design strategies, never guaranteed numbers. Where you see "you should see something like", the exact output on your device may differ.

> **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, data loss, rejected builds, or privacy problems.

Code appears in `monospace`, with the file it belongs to shown above each listing, such as `DropwiseWidgets/TodayProgressWidget.swift`. Long lines are wrapped to fit the page. Menu paths in Xcode are written in bold with arrows, such as **File > New > Target**; labels can differ slightly between Xcode versions. Interface sketches are drawn as text mockups and are original illustrations, not screenshots.

Contents

Preface

PART I — FOUNDATIONS

1 Beyond the App Icon: How iOS Extends Your App into the System

2 Setting Up Dropwise: Targets, Extensions, and Project Structure

3 App Groups and Shared Data: One Store for the App and Every Extension

4 Swift 6 Concurrency Across Processes: Sendable Models, Actors, and Safe Shared State

5 Deep Links and Communication Between the App and Its Extensions

PART II — WIDGETKIT

6 WidgetKit Fundamentals: Entries, Timeline Providers, and Your First Widget

7 Widget Families and Layout: Home Screen, Lock Screen, StandBy, and Tinted Modes

8 Keeping Widgets Fresh: Timelines, Reload Policies, Budgets, and Relevance

9 Interactive Widgets: Buttons and Toggles Powered by App Intents

10 Configurable Widgets: AppIntentConfiguration, Entities, and Dynamic Options

PART III — LIVE ACTIVITIES

11 Live Activities Fundamentals: ActivityKit, Attributes, and Content State

12 Designing for the Lock Screen and the Dynamic Island

13 Updating Live Activities Remotely: Push Tokens, APNs, and Your Server

14 Advanced Live Activities: Push-to-Start, Broadcast Updates, Buttons, and Apple Watch

PART IV — APP INTENTS AND SIRI

15 App Intents Fundamentals: Your First Intents and the Shortcuts App

16 Parameters, Enums, Dialogs, and Results: Designing Intents People Can Use

17 App Entities and Queries: Exposing Dropwise Data to the System

18 App Shortcuts and Siri: Zero-Setup Phrases and Discoverability

19 Spotlight, Apple Intelligence, and Beyond: Indexed Entities, Assistant Schemas, and iOS 26 Additions

PART V — CONTROLS, NOTIFICATIONS, AND SHARING

20 Controls for Control Center, the Lock Screen, and the Action Button

21 Focus Filters: Adapting Dropwise to the User's Focus

22 Rich Notifications: Categories, Actions, and Time-Sensitive Alerts

23 Notification Service and Content Extensions

24 The Share Extension: Bringing Content into Dropwise

PART VI — PRODUCTION

25 Background Work and Budgets: Refresh Tasks and Keeping Every Surface in Sync

26 Testing and Debugging Extensions: Previews, Logs, Schemes, and Unit Tests

27 Privacy, Performance, and Polish: Manifests, Memory Limits, and Accessibility

28 Case Study: Shipping Dropwise Across Every Surface

APPENDICES

A Cheat Sheet: WidgetKit, ActivityKit, and App Intents

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

How iOS lets your app reach beyond its own window, and the groundwork every surface depends on: extension targets and project structure, App Groups and one shared store, Swift 6 concurrency across processes, and deep links that carry people from any surface back into the right screen.

1 · Beyond the App Icon

2 · Setting Up Dropwise

3 · App Groups and Shared Data

4 · Swift 6 Concurrency Across Processes

5 · Deep Links and Communication

CHAPTER 1 · PART I — FOUNDATIONS

Beyond the App Icon: How iOS Extends Your App into the System

Your app's most frequent moments happen outside its own window: a glance at a widget, a Live Activity on the Lock Screen, a request to Siri, a tap in Control Center. This chapter explains how iOS lets an app reach those places, the three ideas that shape every system surface, and the plan for extending Dropwise, our hydration and habit tracker, into all of them.

Your App Is Bigger Than Its Window

Think about how you use the apps on your own phone. You open some of them for long sessions, but for most apps the interaction is measured in seconds. You check a widget, follow a delivery in the Dynamic Island, tap a button in a notification, or say a short phrase to Siri. In every one of those moments, the app's main interface never appears.

For a long time, building an iOS app meant building screens. Today that is only part of the job. iOS offers a family of *system surfaces*, places outside your app's window where your app can show information or accept an action. Each surface has its own framework, its own rules, and its own limits, but they share a common shape: your app prepares something in advance or describes something precisely, and the system decides when and how to present it.

This book is about those surfaces. We take one existing SwiftUI app and extend it, step by step, until it appears everywhere iOS allows. If you are new to SwiftUI, learn the basics first; we teach the model behind each surface, not views and modifiers.

A Tour of the System Surfaces

Before we discuss the ideas that connect them, let us walk through the surfaces one by one. For each, we describe what a person sees, what your code provides, and which part of this book covers it.

Home Screen Widgets

Widgets are the most familiar surface. A person picks one from the widget gallery and sees a glanceable piece of your app every time they unlock. Widgets come in families: small, medium, and large on iPhone, plus an extra-large size on iPad. They are built with WidgetKit and drawn with SwiftUI, but, as we will see shortly, a widget is not a running copy of your app. It is a series of prepared views that the system shows at the times you request.

Since iOS 17, widgets can also be *interactive*: a widget can contain buttons and toggles that perform an action without opening the app. The action is described by an App Intent, which connects widgets to the rest of the system in a way we will explore throughout Part II.

Lock Screen Widgets and StandBy

iOS 16 added widgets to the Lock Screen, in three compact *accessory* families: circular, rectangular, and inline (a single line of text above the clock). These are drawn in a restrained style that adapts to the person's wallpaper, so they rely on shape and contrast rather than your brand colors. The same families are shared with Apple Watch complications.

StandBy, introduced in iOS 17, turns a charging iPhone lying on its side into a bedside or desk display. It shows small Home Screen widgets enlarged, and at night it can render them in a dim, tinted style. You do not build a separate StandBy widget; you make sure your small widget looks good when the system scales it up and changes its rendering mode. Chapter 7 covers families, rendering modes, and StandBy in detail.

Live Activities and the Dynamic Island

A Live Activity shows an event in progress: a ride on its way, a match score, a timer, a delivery. It appears on the Lock Screen and, on iPhone models with the Dynamic Island, in the area around the camera at the top of the screen. Live Activities were introduced in iOS 16.1 and are built with ActivityKit for the data and WidgetKit with SwiftUI for the presentation. They have a clear beginning and end, can be updated from the app or your server, and can contain buttons. Part III covers them.

App Intents: Siri, Shortcuts, Spotlight, and More

App Intents are the connective tissue of the modern system. An App Intent is a Swift type that describes one action your app can perform, with a title, typed parameters, and a `perform()` method. Once declared, the action is available to Shortcuts, Siri, Spotlight, widget and Live Activity buttons, controls, and the Action button. App Shortcuts go one step further and give an intent spoken phrases that work as soon as the app is installed, with no setup by the user.

Intents can also expose your app's data as *entities*, so the system can talk about "the Vitamins habit" rather than a string someone typed. iOS 18 added ways to index those entities in Spotlight and to describe actions for Apple Intelligence. Part IV is devoted to App Intents, and they reappear in almost every other part, because so many surfaces are built on them.

Controls, Control Center, and the Action Button

iOS 18 opened Control Center to third-party apps through *controls*: buttons and toggles that run an App Intent. A person can place your control in Control Center, on the Lock Screen in the slots at the bottom, or, on iPhone models that have one, assign it to the Action button. Controls live in a widget extension: a symbol, a short label, and perhaps an on or off state. Chapter 20 builds two of them.

Focus Filters

Focus lets people silence distractions while they work, sleep, or drive. Since iOS 16, an app can offer a *Focus filter*, an option the person configures in the Focus settings that changes how the app behaves while that Focus is on. For Dropwise, that means pausing drink reminders during a Sleep or Work Focus. A Focus filter is an App Intent too, which shows again how central intents have become. Chapter 21 covers them.

Notifications

Notifications have grown far beyond a line of text. They carry action buttons, group into threads, and use interruption levels. A *notification service extension* can rewrite a remote notification before it is shown, and a *notification content extension* can draw a custom interface when the person expands one. Chapters 22 and 23 cover local reminders, actions, and both extensions.

The Share Sheet

A Share extension puts your app into the share sheet of every other app. It receives a web page or text, shows a small interface, and saves the content for the app. Chapter 24 uses one for saving habit ideas.

Surface	Framework	Introduced	Chapters
Home Screen widgets	WidgetKit	iOS 14 (interactive: iOS 17)	6–10
Lock Screen widgets	WidgetKit	iOS 16	7
StandBy	WidgetKit	iOS 17	7
Live Activities	ActivityKit, WidgetKit	iOS 16.1	11–14
Siri, Shortcuts, Spotlight	App Intents	iOS 16	15–19
Controls	WidgetKit, App Intents	iOS 18	20
Focus filters	App Intents	iOS 16	21
Rich notifications	UserNotifications	long-standing	22–23
Share extension	App extensions	long-standing	24

Three Ideas Behind Every Surface

Each surface has its own API, and learning them one at a time can feel like memorizing a list of unrelated rules. It becomes much easier once you see the three ideas that run through all of them: your code runs in separate processes, the system displays snapshots rather than live views, and everything happens within budgets the system controls. A fourth idea, privacy, shapes what you are allowed to show and where.

Idea 1: Separate Processes

When you add a widget, a notification service, or a Share extension to your project, Xcode creates an *app extension*: a separate bundle that ships inside your app but runs in **its own process**. The system launches that process when it needs it, gives it a small amount of time and memory, and terminates it when it is done. The app may not be running at all, and when both run, they do not share memory. A singleton in your app is a different object from the "same" singleton in your widget extension, and a value you put into `UserDefaults.standard` in the app is invisible to the extension.

This is the source of the most common first bug in extension work. You write a value in the app, read it in the widget, and get nothing.

Dropwise/TodayView.swift (excerpt, the wrong way)

```
// Runs in the app's process. Looks fine, but no extension will ever see it:
// UserDefaults.standard belongs to this process's own sandbox.
UserDefaults.standard.set(1650, forKey: "todayTotal")
```

The fix is an **App Group**: a shared container that the app and each extension can open, enabled by the App Groups capability with the same identifier on every target. Inside it you can keep files and a shared `UserDefaults` suite. Dropwise keeps a single JSON data file there, read and written through one store type that every target links.

Packages/DropwiseKit/Sources/DropwiseKit/DropwiseStore.swift (preview)

```
// Any target that links DropwiseKit and has the App Group capability
// reads and writes the same file in the shared container.
let summary = try await DropwiseStore.shared.logDrink(amountML: 250, beverage: .water)
// summary.totalML == 1650 on the sample day
```

Chapter 3 builds that store, and Chapter 4 explains how Swift 6's concurrency checking helps keep it safe when two processes might touch the same file at the same moment. For now, the key point is simple: **every surface outside the app runs somewhere else, and the only things it knows are what you put in shared storage or what the system hands it**.

Idea 2: Snapshots, Not Live Views

A widget looks like a small piece of a running app, but it is not one. WidgetKit asks your extension for a *timeline*: a list of entries, each with a date and the data to show at that date. Your extension turns each entry into a SwiftUI view, and the system archives the rendered result and displays it at the right time. Between those moments, none of your code runs. The widget cannot scroll, play video, or run arbitrary animations, and a timer you start in the view will never fire, because the view is not alive.

That sounds limiting until you see what it buys. The system can show many widgets without running many processes, and show yours instantly because the result already exists. A few special views help: text that displays a relative time or a countdown, and progress views driven by a date interval, update on screen without new entries because the system knows how to render them.

Live Activities follow the same model with a twist: their content can be replaced by your app or by a push from your server, but the presentation is still a SwiftUI view rendered by the system from data you supply. Controls display a value the system asks you for. Even App Shortcuts are snapshots of a kind: the system reads your phrases and intent descriptions when the app is installed and updated, not when someone speaks. Whenever something surprises you, ask "who renders this, and when does my code actually run?"

Idea 3: Budgets the System Controls

To protect battery life, the system rations work. Widgets receive a refresh budget that the system manages; Apple's documentation describes a frequently viewed widget as typically getting a few dozen refreshes a day, and you should treat that as guidance, not a guarantee. High-priority push updates to Live Activities are budgeted too. Background tasks run when the system decides conditions are right, which may be much later than you asked, or not at all on a given day. Extensions have time limits and memory limits that Apple does not publish as exact numbers.

The practical lesson is to design for uncertainty. A good Dropwise widget does not depend on refreshing at exactly 4:00 PM. Instead, it supplies a timeline with an entry for each hour in advance, so the pace message changes on time even though no code runs. When the person logs a drink in the app, the app asks WidgetKit to reload; when they tap a button inside the widget, the system reloads that widget after the action finishes. In each case we will say which parts are guaranteed and which are up to the system.

> **WARNING**
>
> Never promise in your app's interface, marketing, or support answers that a widget, background task, or reminder will update "every 15 minutes" or "at exactly 9:00". Neither WidgetKit nor BackgroundTasks offers that guarantee, and recent versions of iOS have changed scheduling behavior more than once. Design so that late or skipped refreshes still show correct, if slightly older, information.

Idea 4: Privacy Comes First

Many surfaces are visible when the phone is locked or sitting on a desk. A Lock Screen widget, a Live Activity, or a notification preview can be read by anyone nearby. People control much of this in Settings, and your app shares the responsibility. It should show only what a person would be comfortable with others seeing, mark sensitive content so the system can redact it on a locked device, and ask for permissions at a moment when the reason is obvious.

Extensions also have to respect Apple's privacy rules in their own right. Each target that uses certain APIs, such as `UserDefaults`, declares its reasons in a privacy manifest. Chapter 27 covers this in depth.

Meet Dropwise

Our running example is **Dropwise**, a fictional hydration and habit tracker. The app already exists when this book begins: it is an ordinary SwiftUI app with four screens, and it has never been outside its own window. Here is what it looks like today.

- **Today** shows a large progress ring with the amount logged so far and the daily goal, a line about whether the person is ahead of or behind a steady pace, quick-add buttons for 150, 250, and 500 mL, and a list of today's drinks.
- **History** shows the last seven days as bars, marks the days where the goal was met, and lists the entries for any day you select.
- **Habits** lists three daily habits, each with a symbol, a check button for today, and the current streak: Morning Stretch, Vitamins, and Read 10 Pages.
- **Settings** holds the daily goal, the hours the person is usually awake, the quick-add amounts, the volume unit, and whether reminders are on.

Five beverages count toward the goal on a one-to-one basis: water, tea, coffee, sparkling water, and juice. The goal is a number the person chooses; Dropwise does not judge it. *Pace* compares the total with a steady line from 0 at wake time to the full goal at sleep time and reports "ahead" or "behind" in milliliters.

> **NOTE**
>
> Dropwise is fictional, and its numbers are for demonstration only. The daily goal in every example is a value the user picked, not a recommendation. Nothing in this book is medical or health advice, and your own apps should be just as careful about presenting user-chosen targets as anything more than that.

The Sample Day

To keep every example concrete and every number checkable, the whole book uses the same sample day: **Tuesday, March 10, 2026**. The goal is 2,000 mL, the person is awake from 07:00 to 22:00, and "now" in our examples is **15:30**. By then the log looks like this.

#	Time	Beverage	mL	Running total
1	07:40	water	250	250
2	09:15	coffee	200	450
3	11:30	water	500	950
4	13:05	tea	200	1,150
5	15:20	water	250	1,400

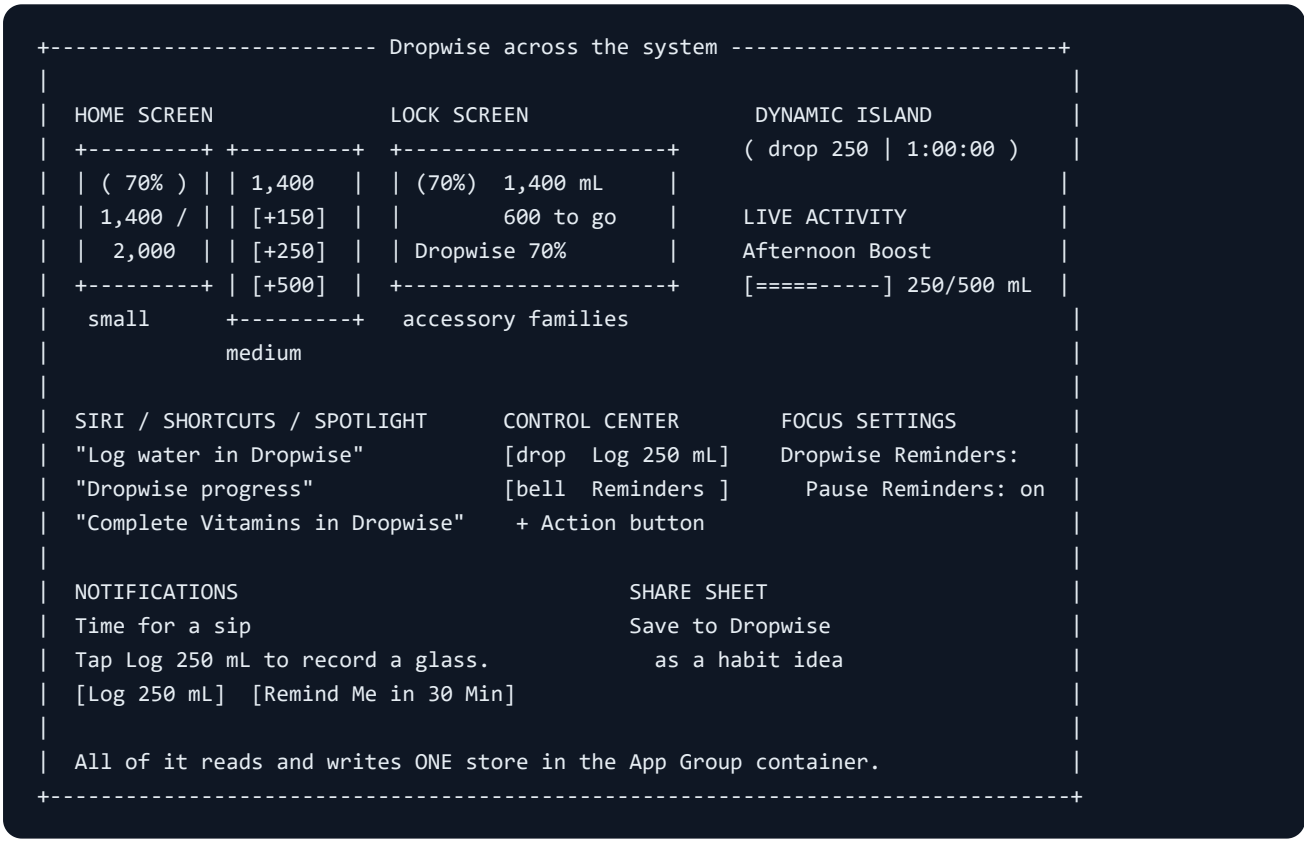
At 15:30 the total is 1,400 mL, which is 70 percent of the goal with 600 mL to go. The steady pace line at 15:30 is 1,133 mL, so the person is 267 mL ahead. The worked example that you will meet in almost every part of the book happens next: at 15:30 the person taps **+250** in the medium Home Screen widget, or asks Siri to log water. The total becomes 1,650 mL, the progress reads 82 percent (we round down), 350 mL remain, and the person is now 517 mL ahead of pace.

Habits on the sample day tell a similar story. Morning Stretch was done at 07:30 and has a 12-day streak. Vitamins have not been taken yet today, and the streak through Monday is 5; marking them done raises it to 6. Read 10 Pages was last done on Sunday, so its streak is 0. A hydration challenge named "Afternoon Boost" (14:30 to 16:30, target 500 mL) later drives our Live Activity.

The Surface Map: Where Dropwise Is Going

Here is the end state we will reach by Chapter 28. The mockup below is a map, not a screen: each box is a surface, and the labels inside show what Dropwise puts there on the sample day at 15:30.

Mockup



Behind the map is a small number of targets. The app target `Dropwise`, a widget extension `DropwiseWidgets` (widgets, Live Activity UI, controls), two notification extensions, a Share extension, and a local package `DropwiseKit` with the models and store that every target links. Chapter 2 creates all of these, and the table below shows where each surface is built.

Surface	What Dropwise shows or does	Built in
Small widget	Progress ring, 70%, opens Today	Chapter 6

Medium widget	Ring plus +150, +250, +500 buttons	Chapters 7, 9
Lock Screen widgets	Gauge, "1,400 mL · 600 to go", "Dropwise 70%"	Chapter 7
Week widget	Seven-day chart, goal met 3 of 6 days	Chapter 7
Habit widget	One chosen habit with a Done button	Chapter 10
Live Activity	Afternoon Boost, 250 of 500 mL, countdown	Chapters 11–14
App Intents and Siri	Log Drink, Today's Progress, Complete Habit	Chapters 15–19
Controls	Log 250 mL button, Reminders toggle	Chapter 20
Focus filter	Pause reminders during a Focus	Chapter 21
Notifications	Reminders with actions, challenge ending	Chapters 22–23
Share extension	Save a link or text as a habit idea	Chapter 24

One Action, Many Surfaces

The most important design decision in the whole project is visible in the surface map if you look closely: the same action, "log a drink", appears in the medium widget, in Siri, in Shortcuts, in Control Center, and in a notification. We do not write five versions of it. We write one App Intent and let each surface invoke it.

Shared/Intents/LogDrinkIntent.swift (preview; Chapter 9 adds it, Chapter 15 explains it)

```
import AppIntents
import DropwiseKit

struct LogDrinkIntent: AppIntent {
    static let title: LocalizedStringResource = "Log Drink"
    static let description = IntentDescription("Adds a drink to today's Dropwise log.")

    @Parameter(title: "Amount (mL)", default: 250, inclusiveRange: (50, 1000))
    var amount: Int

    @Parameter(title: "Beverage", default: .water)
    var beverage: Beverage

    // perform() writes to the shared store and returns the new daily total.
}
```

A widget button creates `LogDrinkIntent(amount: 250, beverage: .water)` and hands it to the system. Siri resolves the same intent from a spoken phrase. A control in Control Center runs it from the Lock Screen. In each case, `perform()` writes to the same store and returns the same result, so the behavior is identical everywhere. Where `perform()` runs differs by surface, a question Chapters 4 and 9 answer.

The second rule follows from the first: **every write reloads every surface**. After a drink is logged from anywhere, the widgets, the controls, and any running Live Activity must catch up. Chapter 5 introduces a small helper for this, and Chapter 25 presents the full matrix of which write triggers which reload.

> **TIP**
> When you plan a new feature for any app, start by listing the *actions* it adds and the *state* it shows, separately. Actions become App Intents, which every surface can share. State lives in shared storage, which every surface can read. Screens come last. That ordering keeps your widget, Siri, and control code from drifting apart.

Versions, Tools, and What You Need

System surfaces change with every major iOS release, so it matters which versions we target. This book makes the following choices.

- **Swift 6 language mode.** All targets compile with Swift 6's complete data-race checking. Models are `Sendable` value types, the shared store is an actor, and UI types are main-actor isolated. Chapter 4 is dedicated to making this work across processes.
- **Xcode 26-era tooling.** Screens, menu names, and template options are described as they appear in recent versions of Xcode. If a name differs slightly on your machine, look for the closest equivalent.
- **Deployment target iOS 18.0.** Every target, including each extension, runs on iOS 18. That gives us interactive and configurable widgets, controls, tinted rendering, push-to-start, and Spotlight entities without availability checks.
- **iOS 26 features are marked.** Where iOS 26 adds something useful, we show it behind an availability check and describe it briefly. The main path always works on iOS 18. Features introduced in iOS 26 are relatively new, so for each one, check the current documentation for details.

Example: an iOS 26 addition behind a check

```
if #available(iOS 26.0, *) {
    // Use the newer API here; Dropwise's main path never depends on it.
} else {
    // The iOS 18 behavior, which every chapter treats as the baseline.
}
```

To follow along you need a Mac with a recent Xcode and an Apple Developer account. Several capabilities we use, such as push notifications, require a paid membership; check Apple's current membership details before you start Part III.

You also want a real iPhone. The Simulator is excellent for widgets, previews, App Intents in Shortcuts, and most notification work, and we use it constantly. Some behavior, however, is only trustworthy on hardware: the Dynamic Island (on iPhone models with the Dynamic Island), the Action button (on models that have one), Siri voice recognition, push delivery timing, Lock Screen privacy behavior, and anything about budgets.

Feature	Simulator	Device recommended for
Widgets and previews	Yes	StandBy, tinted modes, refresh behavior
Live Activities	Yes, local updates	Dynamic Island, push updates, watch
App Intents, Shortcuts	Yes	Siri voice phrases
Controls	Yes	Action button, Lock Screen placement
Notifications	Yes, local	Remote pushes, extensions in real use

How to Read This Book

The book is organized in six parts that follow the order in which a real project would grow. Part I lays the foundation, Part II builds widgets, Part III adds a Live Activity, Part IV covers App Intents and Siri, Part V covers controls, Focus filters, notifications, and sharing, and Part VI makes everything production-ready.

Part I is not optional: every later chapter uses its store and concurrency patterns. After that, you can read Parts II through V in another order if you need one surface urgently.

Conventions Used in This Book

- Every code listing has a caption with its file path and target folder, such as `DropwiseWidgets/TodayProvider.swift`. "(excerpt)" means the listing shows part of a larger file.
- Code uses Swift 6 conventions: `static let` for intent metadata, actors for shared mutable state, `async` and `await` wherever the API allows.
- Runtime behavior is described as "you should see something like…". Console output, refresh moments, and timings vary with devices and iOS releases, so we describe what to look for rather than pretend to show an exact transcript.
- System budgets and scheduling are always hedged. If the text says "the system decides", it means there is no published number and no guarantee.
- Numbers in prose use thousands separators (1,400 mL); numbers in code do not (`1400`).
- Identifiers use a neutral `com.example` prefix and the reserved `.example` domain for Dropwise's fictional server.

Appendix A collects the APIs and limits in a cheat sheet, Appendix B contains copy-ready templates for the files we build, Appendix C is a release checklist, and Appendix D is a glossary. When a chapter refers to "the template", it means the matching listing in Appendix B.

Following One Tap Through the System

At 15:30, the person taps **+250** in the medium widget on the Home Screen. The system launches the widget extension if needed and runs `LogDrinkIntent` there. The intent asks the shared store to add a 250 mL water entry; the store coordinates access to the file in the App Group container, writes the new log, and returns a summary with a total of 1,650 mL. The intent asks the system to reload the other widgets and controls. When `perform()` returns, the system reloads the widget that was tapped, calls the timeline provider, which reads the same file, and archives new views showing 82 percent. If the Afternoon Boost challenge is running, its Live Activity also needs an update, which Part III handles in the app's process. The app's interface never appears, yet every surface shows the same number.

That is the whole book in one paragraph. The chapters that follow explain each step carefully, show the code, and point out the places where it is easy to go wrong.

Try It Yourself

1. Pick an app you use every day and list every surface where it appears outside its window: widgets, Lock Screen, Live Activities, notifications, Siri, Control Center, Focus settings, share sheet. For each one, write down whether it shows state, accepts an action, or both.
2. For one of your own apps, list its three most common actions and its two most important pieces of state. Sketch which surfaces could offer or show each one.
3. Using the sample day, compute by hand the total, percentage (rounded down), and remaining amount after a second tap of +250 at 15:31. Compare your answer with the rule "we round down" and note what the medium widget would show.
4. Look at the surface map and mark every place where the person could see their hydration data while the phone is locked. Decide for each whether to show the exact number, a rough indication, or nothing.
5. Check which iPhone you will use for testing. Find out whether it has the Dynamic Island and the Action button, and note which chapters you will need to verify on another device or in the Simulator.