IOS DEVELOPMENT

iOS Development — Production & Publishing

Real, verified shipping mechanics — production assets, accessibility and localization, Instruments-based performance profiling, app signing and provisioning profiles, TestFlight and Beta App Review, App Store Connect submission and the real privacy/accessibility nutrition labels, App Review Guidelines' own common rejection reasons, StoreKit 2 monetization, and CI/CD with Xcode Cloud and Fastlane — closing with a capstone that carries TaskFlow, this Subject's own recurring app, all the way from finished code to a real, submitted TestFlight build.

10 Chapters | Philip Osztromok | Generated with Claude

CHAPTER 1: PREPARING AN APP FOR PRODUCTION: ASSETS, ICONS & LAUNCH SCREENS

iOS Development — Production & Publishing

Chapter 1 · Preparing an App for Production: Assets, Icons & Launch Screens

Architecture & Data's own capstone left TaskFlow feature-complete but genuinely unpolished — no real icon, no configured launch screen, no real distinct identity of its own. This course starts by turning it into something that looks, and is configured, like a real app ready to ship.

The Asset Catalog: `Assets.xcassets`

Every real Xcode project ships with one central **asset catalog** — a structured real container for icons, images, and colors, managed visually inside Xcode rather than as loose files.

AppIcon

A real, dedicated icon set — the app's own real identity on the Home Screen and throughout the system.

Image Sets

Real images, optionally with distinct 1x/2x/3x variants for different real screen densities.

Color Sets

Named real colors — `Color("AccentColor")` — that can define genuinely different values for light and dark mode.

A Real, Genuinely Simplified App Icon Requirement

Older Xcode versions once required a full real set of icon images at many specific sizes. The current, real requirement is dramatically simpler: a **single 1024×1024 image** in the `AppIcon` set — Xcode itself generates every other real size the system actually needs (Home Screen, Settings, Spotlight, notifications) automatically from that one source image.

A REAL, GENUINE REQUIREMENT ON THE SOURCE IMAGE ITSELF

The real 1024×1024 source image must have **no transparency** — a real, fully opaque background — and no rounded corners of its own; the system applies real corner masking automatically at every displayed size. Submitting an icon with transparency is a real, common App Store Connect rejection reason, covered again once Course 3 reaches submission.

Real Color Sets for Light & Dark Mode

```
Text("TaskFlow")
    .foregroundStyle(Color("BrandAccent"))
```

A real Color Set named `BrandAccent`, defined once in the asset catalog with a distinct value for the "Any Appearance" and "Dark" variants, means `Color("BrandAccent")` automatically resolves to the correct real color depending on the system's own current appearance — no manual `@Environment(\.colorScheme)` branching needed anywhere in the app's own real view code for something this simple.

The Launch Screen: A Real, Storyboard-Free Approach

A launch screen is shown real, briefly, the instant an app starts, before its actual first real view renders — giving the system something immediate to display rather than a blank window. The current, real, recommended approach needs no separate storyboard file at all:

```
<key>UILaunchScreen</key>
<dict>
    <key>UIColorName</key>
    <string>BrandAccent</string>
    <key>UIImageName</key>
    <string>LaunchLogo</string>
</dict>
```

> **A REAL, DELIBERATELY MINIMAL SYSTEM**
>
> A real `UILaunchScreen` dictionary in `Info.plist` — even an empty one, `<dict></dict>` — is enough to satisfy the system's own requirement for something to show at launch. It's deliberately not meant to be a real, animated splash screen or a place for genuine app logic — it exists purely as a brief, real visual bridge before the actual app's own first screen is ready.

Display Name & Bundle Identifier

SETTING	REAL ROLE
`CFBundleDisplayName`	The real, user-visible name shown under the Home Screen icon — can genuinely differ from the Xcode project's own internal name.
Bundle Identifier	A real, reverse-DNS-style unique string (e.g. `com.osztromok.taskflow`) identifying the app to the system and to App Store Connect — genuinely difficult to change once real users have installed it.

SET THIS EARLY, DELIBERATELY

The real bundle identifier is tied to every later step this course covers — signing, TestFlight, App Store Connect all key off it directly. Choosing it carefully now, rather than changing it later, avoids real, genuine rework in every one of those later chapters.

Hands-On Exercises

EXERCISE 1

Add a real 1024×1024 image to TaskFlow's own `AppIcon` set in Xcode, and explain, in your own words, why that source image must have no transparency, connecting your answer to how the system displays it at smaller real sizes.

View solution

EXERCISE 2

Define a real Color Set named `BrandAccent` with a distinct value for light and dark appearance, then use it via `Color("BrandAccent")` in TaskFlow's own `TaskListView` navigation title styling.

View solution

EXERCISE 3

Explain, in your own words, why a real bundle identifier like `com.osztromok.taskflow` is genuinely riskier to change later than most other project settings, once real users have already installed the app.

View solution

CHAPTER 1 QUICK REFERENCE

- `Assets.xcassets` — the real, central home for app icons, images, and colors
- A single, real, opaque 1024×1024 source image is all `AppIcon` needs — Xcode generates every other real size automatically
- Real Color Sets let one named color resolve differently for light vs. dark mode, with zero manual branching in view code
- The real, current launch-screen approach is a storyboard-free `UILaunchScreen` dictionary in `Info.plist`
- `CFBundleDisplayName` (user-visible name) and the bundle identifier (a real, unique, reverse-DNS string) are foundational settings this entire course builds on

CHAPTER 2: ACCESSIBILITY & LOCALIZATION

iOS Development — Production & Publishing

Chapter 2 · Accessibility & Localization

A real app ready to ship needs to genuinely work for users this course hasn't specifically written for yet — VoiceOver users, users who've increased their real system text size, and users in a real, different language. This chapter covers all three, at genuinely low real cost given how much SwiftUI already does by default.

VoiceOver: `.accessibilityLabel()` and `.accessibilityHint()`

A real `Text` view is already read aloud by VoiceOver automatically. An icon-only `Button`, though, has nothing real for VoiceOver to announce on its own:

```
Button {
    isShowingNewTask = true
} label: {
    Image(systemName: "plus.circle.fill")
}
.accessibilityLabel("Add Task")
.accessibilityHint("Opens a form to create a new task")
```

MODIFIER	REAL ROLE
`.accessibilityLabel()`	What VoiceOver actually announces for this element — real, essential for anything with no visible real text of its own.
`.accessibilityHint()`	An optional, real, brief description of what happens after activating it — genuinely supplementary, not a replacement for the label.

A Real, Genuinely Free Default: Dynamic Type

ALREADY WORKING, BY REAL DEFAULT

Every real SwiftUI built-in text style — `.font(.body)`, `.font(.headline)`, `.font(.largeTitle)` — automatically scales with the user's own real Dynamic Type setting, with genuinely zero extra code. Every `Text` view in TaskFlow since Fundamentals Chapter 5 has already had this real, built-in support the whole time.

A REAL, COMMON WAY TO ACCIDENTALLY BREAK IT

`.font(.system(size: 17))` — a real, fixed point size — genuinely does **not** scale with Dynamic Type. Reaching for a real, built-in text style instead of a hardcoded size is what keeps this real accessibility support intact; it isn't automatic once a fixed size is used instead.

Grouping with `.accessibilityElement(children:)`

A `TaskRow` with several separate real subviews (title, priority badge) is, by real default, announced by VoiceOver as several separate real elements — genuinely tedious to navigate one row at a time. Combining them into one real, coherent announcement:

```
HStack {
    Text(task.title)
    Text(priorityLabel)
}
.accessibilityElement(children: .combine)
```

`.combine` merges every real child's own accessibility content into one real, single announcement — "Buy milk, Medium," read once, rather than as two separate, disjointed real stops.

Localization: Xcode's Real String Catalog

Introduced with **Xcode 15, 2023**, the real **String Catalog** (`.xcstrings`) is the current, recommended way to localize an app — replacing the real, older `.strings` / `.stringsdict` file pair.

Real Automatic Extraction

One Real File Per Project

Real, Built-In Pluralization

Xcode scans real, existing `Text("...")` and `String(localized:)` calls and populates the catalog automatically.

Every language's own real translation lives in one `.xcstrings` file, browsable in Xcode's own dedicated editor.

Genuine plural variations (0/1/few/many) are configured per string, no separate `.stringsdict` file needed.

```
Text("Add Task")   // real, automatically picked up into Localizable.xcstrings

let count = tasks.count
Text("\(count) tasks remaining")   // a real, genuine pluralization candidate
```

A REAL, PRACTICAL WORKFLOW

Write real, ordinary English strings directly in SwiftUI code as normal — Xcode's own real "Export Localizations" feature (Product → Export Localizations) generates a real, complete file ready to hand to a translator, and importing their finished work populates every real matching entry in the catalog automatically.

Hands-On Exercises

EXERCISE 1

Add a real `.accessibilityLabel()` and `.accessibilityHint()` to TaskFlow's own icon-only "Add" toolbar button, and explain, in your own words, why an icon-only button genuinely needs a label while a `Text("Add")` button would not.

View solution

EXERCISE 2

Apply `.accessibilityElement(children: .combine)` to TaskFlow's own `TaskRow`, combining its title and priority badge into one real, single VoiceOver announcement.

View solution

EXERCISE 3

Explain, in your own words, why using `.font(.body)` instead of `.font(.system(size: 17))` throughout TaskFlow is what actually determines whether Dynamic Type genuinely works, even though both might display at a similar real size by default.

View solution

CHAPTER 2 QUICK REFERENCE

- `.accessibilityLabel()` / `.accessibilityHint()` — real, essential for any element with no visible real text of its own, like an icon-only button
- Dynamic Type works automatically with every real built-in text style — genuinely broken only by a hardcoded fixed point size
- `.accessibilityElement(children: .combine)` — merges several real subviews into one coherent VoiceOver announcement
- String Catalogs (`.xcstrings`, Xcode 15/2023) — the real, current localization approach, with automatic string extraction and real, built-in pluralization
- Product → Export/Import Localizations — the real, practical workflow for handing strings to a translator and bringing their work back in

CHAPTER 3: PERFORMANCE PROFILING WITH INSTRUMENTS

iOS Development — Production & Publishing

Chapter 3 · Performance Profiling with Instruments

Architecture & Data's own Chapter 7 introduced Instruments through the lens of finding a strong reference cycle. This chapter returns to it with a genuinely different, shipping-focused question: does this real app actually feel fast, responsive, and quick to launch?

The Real SwiftUI Instrument: Counting Re-Renders

A real, dedicated SwiftUI Instrument template measures how often each view's own `body` property is genuinely re-evaluated — a real, direct, measured answer to "is this view redrawing more often than it needs to?"

> **A REAL, CONCRETE SYMPTOM TO WATCH FOR**
>
> A `TaskRow` re-rendering every time *any* task in the list changes — not just the one it's actually displaying — is a real, measurable sign that state is scoped too broadly. Fundamentals Chapter 6's own `@State` / `@Observable` material already explained the mechanism; this instrument is the real, concrete tool for actually catching it happening in practice.

The Real Hangs Instrument: Main-Thread Responsiveness

A real "hang" is any stretch of time the main thread is genuinely blocked long enough that the UI stops responding to touch — the Hangs instrument measures exactly this, flagging real, specific moments the app felt frozen.

> **A REAL, DIRECT CONNECTION TO ARCHITECTURE & DATA CHAPTER 2**
>
> A hang is the real, concrete, user-visible symptom of the exact risk that course's own `@MainActor` chapter warned about — genuine, heavy synchronous work running where it blocks the main thread. Real, expensive work (a large real JSON decode, heavy real image processing) belongs on a background `Task`, never directly inside a `@MainActor` method's own synchronous body.

The Real App Launch Instrument

A real, dedicated App Launch template measures cold-launch time end to end — from the moment a user taps the icon to the moment the first real, meaningful content appears — broken down by real phase (process launch, `UIKit`/SwiftUI setup, the app's own first real code).

INSTRUMENT	REAL, CONCRETE QUESTION IT ANSWERS
SwiftUI	Is any view re-rendering more often than it genuinely needs to?
Hangs	Did the main thread genuinely stop responding, and for how long, and where?
App Launch	How long does the app genuinely take to become usable from a cold start?
Time Profiler / Allocations / Leaks	Real CPU and memory profiling — covered in Architecture & Data, Chapter 7

Custom Instrumentation: `OSSignposter`

For a real, specific operation none of Instruments' own built-in templates directly labels — `syncWithServer()`, say — a real signpost marks it explicitly, showing up as its own named interval directly in Instruments' own timeline.

```
import os

let signposter = OSSignposter(logHandle: OSLog(subsystem: "com.osztromok.taskflow", cate

func syncWithServer() async {
    let interval = signposter.beginInterval("Sync With Server")
    defer { signposter.endInterval("Sync With Server", interval) }

    // the chapter 3 (Architecture & Data) real sync logic, unchanged
}
```

A REAL, GENUINELY PRECISE QUESTION, ANSWERED DIRECTLY

Instead of guessing whether `syncWithServer()` is a genuine contributor to a slow launch, a real signpost interval shows its exact real start and end time laid directly alongside every other real event on the same timeline — a concrete, measured answer, not an inference.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why the SwiftUI Instrument's own real re-render count for `TaskRow` would be a genuinely useful, concrete signal that `@Observable` state (Fundamentals Chapter 6) is scoped too broadly, rather than a purely cosmetic curiosity.

View solution

EXERCISE 2

Add a real `OSSignposter` interval around `TaskListViewModel`'s own `syncWithServer()` method (from Architecture & Data's own capstone), following the chapter's own established pattern.

View solution

EXERCISE 3

Explain, in your own words, why a real hang detected by the Hangs instrument is directly connected to Architecture & Data's own `@MainActor` material, using a concrete example of what kind of real code, if written carelessly, would cause one.

View solution

CHAPTER 3 QUICK REFERENCE

- The SwiftUI Instrument measures how often a view's own `body` genuinely re-renders — a real, concrete tool for catching over-broad state scoping
- The Hangs instrument flags real, specific moments the main thread stopped responding — the concrete, measurable symptom Architecture & Data's own `@MainActor` chapter warned about

- The App Launch instrument measures real cold-launch time, broken down by phase
- `OSSignposter` marks a real, specific operation's own start and end directly on Instruments' own timeline, for questions no built-in template answers directly
- Time Profiler/Allocations/Leaks — already covered in Architecture & Data, Chapter 7 — remain the real, foundational tools underneath all of this

CHAPTER 4: APP SIGNING, PROVISIONING PROFILES & CERTIFICATES

iOS Development — Production & Publishing

Chapter 4 · App Signing, Provisioning Profiles & Certificates

Every chapter so far has run the app straight from Xcode with almost no thought given to why it was even allowed onto a real device in the first place. This chapter opens that up directly — the real machinery of certificates and provisioning profiles that has been working quietly in the background the entire course, and that becomes unavoidable the moment an app needs to leave a single development Mac.

Why Code Signing Exists at All

iOS refuses to run any binary that isn't signed by a real, trusted identity — this is the real, fundamental reason a random unsigned app can't simply be copied onto an iPhone and launched. Code signing exists to answer one specific question before the OS ever runs a line of code: who, exactly, is vouching for this binary?

Certificates: A Real Public/Private Key Pair

A signing certificate is built on a real public/private key pair. The private key stays on the Mac that created it, held securely in Keychain — it never leaves, and it's what actually signs the app binary. The public key is the half embedded in a provisioning profile, letting iOS verify that a signature genuinely came from the identity it claims to.

Apple's own current certificate types, verified directly against Apple's real certificates-overview documentation:

CERTIFICATE	OWNED BY	REAL, CONCRETE PURPOSE
Apple Development	An individual (per-Mac)	Run and debug an app on real, registered devices during development
Apple Distribution	The team as a whole	Distribute the app for testing or submit it to App Store Connect

A REAL, VERIFIED DETAIL

An Apple Development certificate genuinely appends the Mac's own computer name to its identity — a real certificate might literally be named something like `"Philip Osztromok (MacBook Pro)"`. An Apple Distribution certificate, by contrast, belongs to the whole team rather than one machine, and only an Account Holder or Admin can create one — a real, verified permission restriction, not an arbitrary convention.

What a Provisioning Profile Actually Binds Together

A provisioning profile is the real bridge connecting every other piece of code signing into one package. It contains:

App ID

The real, unique bundle identifier this profile is valid for — the same identifier Chapter 1's own app-icon/launch-screen setup already relied on

Certificate (public key)

Lets iOS verify the app's own signature was really produced by the matching private key

Entitlements

The real capabilities the app is permitted to use — directly tied to Architecture & Data Chapter 8's own camera/location/notification permissions

Devices

For development and ad hoc profiles: the real, specific device UDIDs the app is allowed to run on

The Four Real Provisioning Profile Types

PROFILE TYPE	REAL DEVICE LIMIT	REAL USE CASE
Development	Specific registered devices	Testing on the developer's own real devices
Ad Hoc	Specific registered devices (max 100)	Beta testing with a small, named group
App Store	No device limit	Public release via the App Store

PROFILE TYPE	REAL DEVICE LIMIT	REAL USE CASE
Enterprise	No device limit	Large-scale internal distribution

A REAL, CONCRETE DISTINCTION WORTH REMEMBERING

Development and Ad Hoc profiles are genuinely limited to a specific, named list of real devices — App Store and Enterprise profiles carry no such limit at all. This is exactly why an Ad Hoc build sent to a beta tester's iPhone that was never actually registered will refuse to install, no matter how correctly everything else was signed.

Automatic vs. Manual Signing in Xcode

Xcode's own real "Automatic" signing mode creates and renews certificates and provisioning profiles on the developer's behalf whenever it detects they're missing or expired — the mode every earlier chapter in this course has quietly relied on. "Manual" signing instead requires selecting a specific, already-created certificate and profile directly, giving full control at the real cost of managing renewal by hand.

The Real Apple Developer Program Membership

None of this is possible without a real, paid Apple Developer Program membership — verified directly against Apple's own current program page at **$99/year**, which unlocks App Store distribution, TestFlight, and code-level support. A separate real Apple Developer Enterprise Program, for private internal distribution to a company's own employees, costs **$299/year**.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why a provisioning profile embeds only the certificate's own public key rather than the private key, and what would go wrong if the private key were included instead.

View solution

EXERCISE 2

A beta tester reports the app refuses to install on their own real iPhone, even though the build was signed correctly. Using the chapter's own real device-limit distinction, explain the single most likely cause and how to fix it.

View solution

EXERCISE 3

Explain, in your own words, why the Push Notification capability from Architecture & Data Chapter 8 requires an entitlement inside the app's own provisioning profile, connecting the two chapters directly.

View solution

CHAPTER 4 QUICK REFERENCE

- A certificate is a real public/private key pair — the private key stays on the Mac, the public key goes into the provisioning profile
- **Apple Development** — individual, per-Mac; **Apple Distribution** — team-owned, one per team
- A provisioning profile binds together the App ID, certificate, entitlements, and (for Development/Ad Hoc) a real list of registered devices
- Development and Ad Hoc profiles are device-limited (Ad Hoc: max 100); App Store and Enterprise are not
- Xcode's Automatic signing manages certificates and profiles for you; Manual signing requires choosing them directly
- Apple Developer Program: real $99/year; Enterprise Program: real $299/year

CHAPTER 5: TESTFLIGHT & BETA DISTRIBUTION

iOS Development — Production & Publishing

Chapter 5 · TestFlight & Beta Distribution

Chapter 4's own Ad Hoc profile — device-limited, up to 100 registered UDIDs — is the mechanism real beta distribution used to run on before Apple built a dedicated tool for it. TestFlight is that dedicated tool: Apple's own real, official beta-testing platform, built specifically to remove the manual UDID/provisioning-profile burden Chapter 4 described.

Internal vs. External Testers: A Real, Verified Distinction

TestFlight draws a real, sharp line between two genuinely different kinds of tester, verified directly against Apple's own TestFlight documentation:

	INTERNAL TESTERS	EXTERNAL TESTERS
Who	Up to 100 members of the actual development team	Up to 10,000 testers outside the team
Requires App Review?	No	**Yes** — Beta App Review
Real roles required	Account Holder, Admin, App Manager, Developer, or Marketing	None — any invited person
Invitation method	Added directly to the team on App Store Connect	Individual email invite, or a real public link

> **A REAL, VERIFIED DETAIL WORTH NOTICING**
>
> Internal testers can be set to automatically receive every new build — no manual re-invitation needed. External testers cannot skip Beta App Review even for a build that's already been through it once before: a real, new build submitted to an existing external test group is verified to be automatically sent for review again before it reaches those testers.

Beta App Review: What It Actually Checks

Beta App Review exists specifically to gate the moment a build first reaches anyone outside the actual development team. The real requirement, verified directly: the very first build offered to external testers must already be approved before those testers can install it — a genuine, real checkpoint, distinct from (and lighter-weight than) the full App Store review Chapter 6 covers next.

Build Expiration

A TestFlight build doesn't stay installable forever — it expires after a real, fixed window, after which testers on that build must update to a newer one to keep using the app at all. This is a deliberate, real safeguard: it keeps testers from running an old, unmaintained build indefinitely once a project has moved on.

> **WHY THIS MATTERS IN PRACTICE**
>
> A beta test that goes quiet for a couple of months genuinely risks every tester's own build silently expiring in the background — a real, common source of "the app just stopped working" reports that has nothing to do with a bug in the code at all.

Getting a Build Into TestFlight

A build reaches TestFlight through the exact same real Archive step Chapter 4's own signing machinery underpins — Xcode builds and signs the app using a real Apple Distribution certificate, then uploads it directly to App Store Connect.

```
// Product → Archive in Xcode, then, from the Organizer window:
// 1. Select the new archive
// 2. Distribute App → App Store Connect → Upload
// 3. Xcode signs the build with the real Apple Distribution certificate
//    from Chapter 4, using the matching App Store provisioning profile
// 4. The build appears in App Store Connect's TestFlight tab
//    once Apple finishes real, automated processing
```

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why an internal tester never triggers Beta App Review while an external tester always does, connecting this back to the real trust boundary the review exists to protect.

View solution

EXERCISE 2

A team wants to invite 500 outside beta testers using a public link rather than 500 individual email invitations. Explain whether this is possible under TestFlight's own real tester categories, and why.

View solution

EXERCISE 3

Explain, in your own words, how Chapter 4's own Ad Hoc provisioning profile and TestFlight solve genuinely the same real problem (getting a build onto a real device outside the developer's own Mac) using two different mechanisms — and what real limitation of the Ad Hoc approach TestFlight removes.

View solution

CHAPTER 5 QUICK REFERENCE

- **Internal testers** — up to 100 team members, no App Review, automatic new-build delivery
- **External testers** — up to 10,000 testers, real Beta App Review required, invited by email or a real public link
- Beta App Review gates the first build any external tester ever sees, and re-triggers on every new build sent to an existing external group
- A TestFlight build expires after a real, fixed window — testers must update to keep using the app
- A build reaches TestFlight via Xcode's own Archive → Distribute App → App Store Connect flow, signed with the Chapter 4 Apple Distribution certificate

CHAPTER 6: APP STORE CONNECT & SUBMISSION

iOS Development — Production & Publishing

Chapter 6 · App Store Connect & Submission

A build that's already survived Chapter 5's own TestFlight round is genuinely close to done — but a TestFlight build and a real, public App Store listing are two different things. This chapter covers the real, remaining work: turning that tested build into an actual App Store Connect app record, complete with metadata, and submitting it for the real, full review.

Creating the App Record

Before any build can be uploaded for public release, App Store Connect needs a real app record — created directly from the same Bundle ID that Chapter 4's own App ID material already established, plus a unique SKU (an internal identifier only the developer ever sees) and the app's own public-facing name.

Required Metadata — Verified Against Apple's Own Documentation

App Name & Icon

The real, public identity shown on the App Store — the same real 1024×1024 icon source Chapter 1 already prepared

Description & Subtitle

What the app actually does, plus a short, real promotional subtitle shown near the name

Screenshots & App Previews

Real device screenshots and optional short video previews of the app in use

Keywords

Real search terms that determine what a user typing into the App Store's own search bar will find

Age Rating

A real, structured questionnaire about the app's own content, used to assign the correct real age category

Two Real Nutrition Labels

Verified directly against Apple's own App Store submission documentation, every App Store listing carries two real, structured disclosure labels — genuinely different in scope and requirement:

	PRIVACY NUTRITION LABEL	ACCESSIBILITY NUTRITION LABEL
Requirement	**Mandatory** for every submission	Optional
What it discloses	Every real data-collection practice, including third-party code the app integrates	Real, specific accessibility features the app supports

A REAL, DIRECT PAYOFF OF CHAPTER 2

The Accessibility Nutrition Label is a real, verified App Store Connect field for disclosing exactly the kind of work Chapter 2 covered — VoiceOver support, Voice Control, larger text, captions. Every genuine accessibility improvement made back in Chapter 2 becomes something that can now actually be declared, visibly, on the app's own real public App Store listing.

A REAL, EASY MISTAKE

The Privacy Nutrition Label must genuinely account for third-party code, not just first-party code written directly by the app's own developer — a real analytics SDK or ad framework bundled into the app still counts as a real data-collection practice that has to be disclosed, even though none of that specific code was written in-house.

Submitting for Review

With metadata complete and a real, already-TestFlight-tested build attached, the app is submitted for App Review — the real, full review every public App Store listing must pass, distinct from Chapter 5's own lighter-weight Beta App Review.

A REAL, PRACTICAL TIMING NOTE

Apple has generally guided developers to expect most review decisions within roughly a day or two, though the real, actual duration for any specific submission can vary — a genuinely useful reason to submit with real lead time before any hard external launch date, rather than assuming a same-day turnaround.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why the Privacy Nutrition Label must disclose third-party code's data practices, not just the app's own first-party code.

View solution

EXERCISE 2

Explain, in your own words, how the Accessibility Nutrition Label directly builds on the real work described in Chapter 2, and why declaring it is genuinely different from simply having implemented it.

View solution

EXERCISE 3

Explain, in your own words, why a real hard external launch date is genuinely risky to plan around a same-day App Review turnaround, and what a safer real approach looks like.

View solution

CHAPTER 6 QUICK REFERENCE

- An app record needs the real Bundle ID from Chapter 4, a unique SKU, and a public-facing name
- Required metadata: name, icon, description, subtitle, screenshots, app previews, keywords, age rating
- **Privacy Nutrition Label** — mandatory, must include third-party code's own data practices
- **Accessibility Nutrition Label** — optional, a real, direct payoff of Chapter 2's own accessibility work
- App Review is the real, full review — heavier than Chapter 5's own Beta App Review, with a genuinely variable real timeline

CHAPTER 7: APP REVIEW GUIDELINES: COMMON REJECTION REASONS

iOS Development — Production & Publishing

Chapter 7 · App Review Guidelines: Common Rejection Reasons

Chapter 6 covered how to submit an app for App Review. This chapter covers, in real, specific, guideline-numbered detail, exactly why most real submissions actually get rejected — verified directly against Apple's own current App Review Guidelines page.

The Five Real, Top-Level Guideline Sections

Safety
Objectionable content, user-generated content moderation, and data-privacy protections

Performance
App completeness, real device compatibility, and genuine software stability

Business
Payments, subscriptions, and legitimate acquisition of the app itself

Design
Real design quality and minimum functionality expectations

Legal
Real, applicable privacy law and intellectual property requirements

Rejection Reason 1: App Completeness (Guideline 2.1(a))

Apple's own real, quoted guideline text:

> *"Submissions to App Review... should be final versions with all necessary metadata and fully functional URLs included; placeholder text, empty websites, and other temporary content should be scrubbed before submission."*

Real, specific triggers verified against this guideline:

- Crashes or obvious technical problems during real review
- Placeholder text, empty websites, or temporary content left in

- No demo account provided for a login-required feature
- A backend service that isn't actually enabled or reachable during review

> **A REAL, DIRECT CONNECTION TO THIS COURSE'S OWN CAPSTONE**
>
> A real, login-gated app — TaskFlow, this course's own recurring capstone project, is exactly this shape — genuinely needs a working demo account handed to the reviewer directly, since a reviewer with no way to log in cannot verify the app actually works at all.

Rejection Reason 2: Incomplete In-App Purchases (Guideline 2.1(b))

> *"If you offer in-app purchases in your app, make sure they are complete, up-to-date, visible to the reviewer and functional."*

Rejection Reason 3: Minimum Functionality (Guideline 4.2)

> *"Your app should include features, content, and UI that elevate it beyond a repackaged website. If your app is not particularly useful, unique, or 'app-like,' it doesn't belong on the App Store."*

Real, common examples verified against this guideline:

- An app that's essentially just one song, movie, or book (those belong on iTunes/Apple Books instead)
- A simple link collection or web clipping with no real native functionality
- A template-generated app with no genuine customization

Rejection Reason 4: Metadata Accuracy (Guideline 2.3)

> *"Customers should know what they're getting when they download or buy your app, so make sure all your app metadata... accurately reflect the app's core experience."*

REAL SUB-GUIDELINE	WHAT IT ACTUALLY REQUIRES
2.3.1a	Every new feature must be described specifically in the review notes — hidden/undocumented features are a real, specific rejection trigger
2.3.3	Screenshots must genuinely show the app in use — not just title art or a splash screen
2.3.6	Age ratings must be accurate — a mismatch can trigger real regulatory follow-up

A REAL, DIRECT CONNECTION TO CHAPTER 6

Every real metadata field Chapter 6 covered — description, screenshots, age rating — is exactly what Guideline 2.3 checks for accuracy. Filling those fields out correctly in App Store Connect and filling them out *accurately* are two genuinely different things, and only the second one actually satisfies this guideline.

Rejection Reason 5: Missing Developer Contact Information (Guideline 1.5)

> *"People need to know how to reach you with questions and support issues. Make sure your app and its Support URL include an easy way to contact you."*

Responding to a Rejection

App Store Connect provides a real, direct way to respond to a rejection message from inside the Resolution Center, addressing the reviewer's own stated concern before resubmitting — a genuinely faster path than simply guessing at a fix and resubmitting blind.

Hands-On Exercises

EXERCISE 1

TaskFlow requires a real user account to view any tasks at all. Explain, in your own words, what a submission needs to include to satisfy Guideline 2.1(a) for this specific app, and why omitting it would genuinely block review.

View solution

EXERCISE 2

Explain, in your own words, why a screenshot showing only the app's own launch/splash screen would violate Guideline 2.3.3, and what a genuinely compliant screenshot would show instead.

View solution

EXERCISE 3

Explain, in your own words, why Guideline 4.2's own real "elevate it beyond a repackaged website" standard exists, and give a concrete example of a real, genuinely native feature that would help a simple content-browsing app pass it.

View solution

CHAPTER 7 QUICK REFERENCE

- Five real top-level sections: Safety, Performance, Business, Design, Legal
- **2.1(a)** — app completeness: no placeholders, working demo accounts required for login-gated features
- **2.1(b)** — in-app purchases must be complete, visible, and functional for review
- **4.2** — minimum functionality: must be genuinely "app-like," not a repackaged website
- **2.3** (with 2.3.1a, 2.3.3, 2.3.6) — metadata must accurately reflect the real app
- **1.5** — real, working developer contact information is required
- A rejection can be responded to directly in App Store Connect's own Resolution Center before resubmitting

CHAPTER 8: MONETIZATION: IN-APP PURCHASES & SUBSCRIPTIONS

iOS Development — Production & Publishing

Chapter 8 · Monetization: In-App Purchases & Subscriptions

Chapter 6 mentioned in-app purchases briefly, as one more piece of App Store Connect metadata. This chapter covers the real, working mechanics behind them — StoreKit 2, Apple's modern, Swift-native purchasing API.

The Four Real Product Types

TYPE	REAL, CONCRETE EXAMPLE
Consumable	Purchasable repeatedly — an in-app currency, extra credits
Non-Consumable	A one-time unlock — removing ads, an unlocked premium feature
Auto-Renewable Subscription	Recurring access that renews automatically until cancelled
Non-Renewing Subscription	A fixed-length access period the user must manually repurchase

Making and Verifying a Purchase

StoreKit 2's own real API is genuinely built around Swift's `async` / `await` — the same real concurrency model Architecture & Data Chapter 2 already covered — rather than the older, delegate-callback-based approach of StoreKit 1.

```
let result = try await product.purchase()

switch result {
case .success(let verification):
    let transaction = try checkVerified(verification)
    await transaction.finish()
    // grant real entitlement here
case .userCancelled:
    // real, expected outcome — no error
case .pending:
```

```
    // e.g. real Ask to Buy parental approval still pending
@unknown default:
    break
}
```

A REAL, VERIFIED SECURITY DETAIL

Verified directly against Apple's own StoreKit documentation: every real transaction StoreKit 2 hands back is cryptographically signed by the App Store itself, using JSON Web Signature (JWS) format — the real reason a purchase result can be verified as genuine rather than simply trusted at face value.

Real Commission Rates

Verified directly against Apple's own current Small Business Program documentation:

SITUATION	REAL, VERIFIED COMMISSION
Standard rate	30% of proceeds
Small Business Program	**15%** — real eligibility: under $1 million USD in prior-year proceeds
EU, alternative terms, subscriptions after year one	A real, further-reduced 10%

A REAL, VERIFIED ELIGIBILITY DETAIL

The Small Business Program's real $1 million threshold is calculated on proceeds net of Apple's own commission and certain taxes — and every associated developer account has to be declared, with their combined proceeds counted toward that same real threshold together, not each account counted separately.

Testing Purchases Locally

Verified directly against Apple's own StoreKit documentation: StoreKit Testing in Xcode provides a real local sandbox — a StoreKit Configuration file defining test products directly

inside Xcode, letting purchases be tested entirely on a simulator with zero real App Store Connect setup or real money involved.

Hands-On Exercises

EXERCISE 1

TaskFlow is considering a "Pro" tier unlocking unlimited tasks permanently, versus a recurring monthly cloud-sync subscription. Explain, in your own words, which real StoreKit 2 product type fits each, and why.

View solution

EXERCISE 2

Explain, in your own words, why `.pending` is a genuinely different real outcome from an error, and why an app must handle it as its own distinct case rather than treating it as a failed purchase.

View solution

EXERCISE 3

Explain, in your own words, why testing purchases through StoreKit Testing in Xcode's own local sandbox is genuinely preferable to testing directly against a live App Store Connect in-app purchase during development.

View solution

CHAPTER 8 QUICK REFERENCE

- Four real product types: consumable, non-consumable, auto-renewable subscription, non-renewing subscription
- `product.purchase()` is a real, async StoreKit 2 call — handle `.success`, `.userCancelled`, and `.pending` as genuinely distinct outcomes
- Every real transaction is cryptographically signed (JWS) by the App Store itself

- Standard commission: 30%; Small Business Program (real, verified): **15%** under $1M/year; EU alternative-terms subscriptions after year one: a real 10%
- StoreKit Testing in Xcode provides a real local sandbox with zero App Store Connect setup or real money required

CHAPTER 9: CI/CD FOR IOS (XCODE CLOUD & FASTLANE)

iOS Development — Production & Publishing

Chapter 9 · CI/CD for iOS (Xcode Cloud & Fastlane)

Every real step covered so far — archiving, signing, TestFlight, App Store Connect — has been done by hand, one real click at a time. This chapter covers the two real, established ways to automate that entire pipeline: Xcode Cloud, Apple's own real, native CI/CD service, and Fastlane, the real, open-source automation tool most of the wider iOS community reaches for.

Xcode Cloud: Apple's Own Real CI/CD Service

Verified directly against Apple's own Xcode Cloud documentation: workflows can be configured to automatically build every real committed code change, run tests in parallel across real device configurations, and alert the team the moment a change genuinely introduces a problem — all directly integrated into both Xcode itself and App Store Connect's own web dashboard.

> **A REAL, VERIFIED SECURITY DETAIL**
>
> Every Xcode Cloud build runs inside a real, ephemeral environment — genuinely destroyed once the build completes — with source code accessed only for the build itself, and all data encrypted at rest.

Real, Verified Xcode Cloud Pricing

TIER	REAL, VERIFIED MONTHLY COST
25 hours	Free — included with Apple Developer Program membership
100 hours	$49.99
250 hours	$99.99
1,000 hours	$399.99
10,000 hours	$3,999.99

A real compute hour is the actual time spent executing a task in the cloud — building, testing, archiving. Apple's own real example: running 5 real tests of 12 minutes each together consumes 1 compute hour, since tests genuinely run in parallel rather than one after another.

Fastlane: Real, Open-Source Automation

Verified directly against Fastlane's own site: Fastlane is a real, free, open-source automation platform, governed by the Mobile Native Foundation, with over 700 real contributors. It runs on any CI system — Jenkins, GitHub Actions, GitLab, CircleCI — rather than being tied to one, native, Apple-only pipeline the way Xcode Cloud is.

Its own real, central concept is a **lane** — a named, configured sequence of steps run with a single real command.

```
# Fastfile
lane :beta do
  increment_build_number
  build_app
  upload_to_testflight
end
```

Fastlane's Real Actions — Tied Directly to Earlier Chapters

match

Manages Chapter 4's own real certificates and provisioning profiles, stored in an encrypted git repository the whole team can share

pilot

Automates Chapter 5's own real TestFlight build upload and tester distribution

deliver

Uploads Chapter 6's own real App Store Connect metadata — screenshots, description, and the build itself — for submission

Choosing Between Them

A REAL, PRACTICAL WAY TO DECIDE

Xcode Cloud is genuinely the simpler real starting point — no separate account, no scripting, tightly integrated directly into Xcode and App Store Connect. Fastlane genuinely earns its own extra setup complexity once a real project needs to run the exact same pipeline across multiple CI providers, or needs finer-grained scripted control than Xcode Cloud's own workflow editor offers.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why a compute hour is measured by actual parallel execution time rather than simply adding up every individual test's own duration.

View solution

EXERCISE 2

Explain, in your own words, why Fastlane's `match` action storing certificates and provisioning profiles in an encrypted git repository is genuinely useful for a real team, connecting this back to Chapter 4's own real certificate material.

View solution

EXERCISE 3

A team already runs their Android app's CI/CD entirely through GitHub Actions, and wants their new iOS app's pipeline to live in the exact same GitHub Actions workflow file. Explain, in your own words, which of the two real tools this chapter covers genuinely fits that requirement, and why.

View solution

CHAPTER 9 QUICK REFERENCE

- **Xcode Cloud** — Apple's own real, native CI/CD, tightly integrated into Xcode and App Store Connect; real ephemeral, encrypted build environments

- Real pricing: 25 free hours/month with Apple Developer Program membership, then tiered plans from $49.99 to $3,999.99/month
- **Fastlane** — real, free, open-source, Mobile Native Foundation-governed; runs on any CI provider
- A real **lane** is a named sequence of automated steps run with one command
- `match` (Chapter 4 signing), `pilot` (Chapter 5 TestFlight), `deliver` (Chapter 6 App Store Connect) — Fastlane's own real, named actions for exactly the manual work this course has covered by hand

CHAPTER 10: CAPSTONE — SHIPPING AN APP TO TESTFLIGHT

iOS Development — Production & Publishing

Chapter 10 · Capstone: Shipping an App to TestFlight

TaskFlow has been built, architected, tested, and evolved across all three courses of this Subject. This capstone takes that finished app through the one real, remaining step: shipping it, end to end, from finished code to a real, submitted TestFlight build ready for external testers — walking through every prior chapter's own real contribution in the order it actually happens.

Step 1: Production Assets & Localization (Chapters 1–2)

STEP 1

A real 1024×1024 app icon source is dropped into the asset catalog; the `UILaunchScreen` dictionary is confirmed in `Info.plist`. Every user-facing string — including the ones added throughout Architecture & Data's own capstone — lives in a real String Catalog (`Localizable.xcstrings`), verified against Chapter 2's own automatic-extraction feature. Every real accessibility label and combined `TaskRow` element from Chapter 2 is confirmed still present.

Step 2: A Real Instruments Pass (Chapter 3)

STEP 2

The SwiftUI Instrument confirms `TaskRow` re-renders only when its own specific task changes. The Hangs instrument records zero real hangs across a full test session. A real `OSSignposter` interval, added directly around `syncWithServer()` per Chapter 3's own established pattern, confirms the real sync duration stays well within an acceptable window.

Step 3: Signing With a Real Apple Distribution Certificate (Chapter 4)

STEP 3

A real Apple Distribution certificate is created for the team, and an App Store provisioning profile is generated binding it to TaskFlow's own real Bundle ID and its Push Notifications entitlement from Architecture & Data Chapter 8. Xcode's own Automatic signing mode handles the day-to-day renewal.

Step 4: Internal, Then External TestFlight (Chapter 5)

STEP 4

The signed build is archived and uploaded to App Store Connect. It's first offered to the real, up-to-100-member internal test group — no review required — for a fast first pass. Once confirmed stable, it's added to a real external test group; Beta App Review runs automatically before those testers ever see it, exactly as Chapter 5's own real distinction described.

Step 5: The App Store Connect Record (Chapter 6)

STEP 5

A real app record is created from the same Bundle ID, with name, icon, description, screenshots, and age rating filled in. Both real nutrition labels are completed: the mandatory Privacy label, honestly disclosing the real data practices of any third-party code bundled in, and the optional Accessibility label, declaring the real VoiceOver and Dynamic Type support built in Chapter 2.

Step 6: A Pre-Flight Check Against Common Rejections (Chapter 7)

STEP 6

Before submission, the build is checked directly against Chapter 7's own real, guideline-numbered list: a real demo account is included in the review notes for TaskFlow's own login requirement (2.1(a)); every screenshot shows the app genuinely in use, never a launch screen (2.3.3); the app's own real offline SwiftData caching and push notifications are named as the concrete, native features that satisfy Guideline 4.2's "beyond a repackaged website" standard.

Step 7: A Real StoreKit Pro Tier (Chapter 8)

STEP 7

A real non-consumable "TaskFlow Pro" product is configured in App Store Connect, matching Exercise 1's own reasoning from Chapter 8. The purchase flow — `product.purchase()`, switching on `.success` / `.userCancelled` / `.pending` — is fully tested first through Chapter 8's own local StoreKit Testing sandbox, with zero real money or live App Store systems involved, before ever touching a live purchase.

Step 8: Automating the Pipeline (Chapter 9)

Fastfile

```
lane :release_candidate do
  match(type: "appstore")          # real Ch.4 signing, shared across the team
  increment_build_number
  build_app
  upload_to_testflight(groups: ["Internal"])  # real Ch.5 internal group first
end

lane :submit do
  deliver(submit_for_review: true)  # real Ch.6 metadata + Ch.7-checked submission
end
```

`release_candidate` reuses `match` for Step 3's own signing and `pilot`'s underlying mechanics for Step 4's own internal TestFlight upload. `submit` reuses `deliver` for Step 5's own metadata, submitting only once Step 6's own pre-flight checklist has genuinely passed.

Chapter Attribution

CAPSTONE STEP	CHAPTER
Production assets & localization	Ch.1–2
Instruments performance pass	Ch.3
Apple Distribution signing	Ch.4
Internal → external TestFlight	Ch.5
App Store Connect record & nutrition labels	Ch.6
Pre-flight rejection-reason check	Ch.7
Real StoreKit Pro tier	Ch.8
Fastlane pipeline automation	Ch.9

COURSE COMPLETE — AND THE FULL IOS DEVELOPMENT SUBJECT

This closes iOS Development — Production & Publishing at 10/10 chapters, and with it, the entire three-course **iOS Development** Subject — Fundamentals, Architecture & Data, and Production & Publishing — at 30 chapters total. TaskFlow itself has now been built (Fundamentals), architected and tested (Architecture & Data), and genuinely shipped (this course), the same real app carried end to end across all three.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why Step 4's own internal-then-external TestFlight sequence is a genuinely safer real order than uploading directly to an external test group first.

View solution

EXERCISE 2

Explain, in your own words, why Step 7's own real StoreKit Testing sandbox pass has to happen before the Pro tier product is ever tested through a live purchase, connecting this back to Chapter 8's own real reasoning.

View solution

EXERCISE 3

Explain, in your own words, why the `submit` lane's own `submit_for_review: true` step should only ever run after Step 6's own pre-flight checklist genuinely passes, and what real risk skipping that order introduces.

View solution

CHAPTER 10 QUICK REFERENCE — THE FULL SHIPPING PIPELINE

- Production assets & localization (Ch.1–2) → a real Instruments pass (Ch.3) → Apple Distribution signing (Ch.4)
- Internal, then external TestFlight (Ch.5) → a complete App Store Connect record with both nutrition labels (Ch.6)
- A pre-flight check against real, common rejection reasons (Ch.7) → a real, sandbox-tested StoreKit Pro tier (Ch.8)
- The whole pipeline automated through real Fastlane lanes (Ch.9)
- This closes the full, three-course, 30-chapter iOS Development Subject