Build macOS Apps with SwiftUI Course

Flavio Copes

Updated August 3, 2026

Contents

Build native Mac apps with SwiftUI windows, sidebars, menus, settings, local files, Keychain, menu bar extras, and reliable process integration.

What you'll learn: Build and test a focused native Mac utility with deliberate state ownership, desktop behavior, durable data, and one useful system integration.

Start a macOS app

Create a native target, understand scenes, separate model state, and test real windows.

Create a macOS SwiftUI project

Create the correct Xcode target, name the product deliberately, and verify the app launches as a native Mac application.

You need Xcode to build Mac apps. Download it from the Mac App Store, open it, and choose File → New → Project. Pick the **macOS** tab, then the **App** template. This choice matters more than it looks: the iOS template produces a different target with different capabilities, and converting later is tedious work.

On the options screen, choose SwiftUI for the interface and Swift for the language. Name the product something you will not regret. We will build a notes app throughout this course, so `Notes` works well.

The **organization identifier** deserves a moment of attention. Xcode combines it with the product name to build the **bundle identifier**, something like `com.flaviocopes.Notes`. macOS uses this string as the identity of your app: preferences, Keychain items, and notifications are all tied to it. Use a reverse-DNS name for a domain you control, and treat it as permanent.

Before writing any code, press Cmd+R and run the untouched template. You should see a resizable window with "Hello, world!" in the middle, and the app name in the menu bar next to the Apple menu. This is your working baseline. If something breaks later, you know it was your change, not the toolchain.

Xcode generated two Swift files. The entry point looks like this:

```
import SwiftUI

@main
struct NotesApp: App {
  var body: some Scene {
    WindowGroup {
      ContentView()
    }
  }
}
```

We will unpack every line of this in the next lesson. For now, notice how small it is. There is no storyboard, no window setup code, no boilerplate delegate. SwiftUI apps on macOS start from a declaration.

Open the target settings — click the project in the navigator, then the `Notes` target — and look at the General tab. Record the bundle identifier, the **deployment target** (set it to macOS 14 for this course), and the version and build numbers. These values follow the app through its whole life, from the first debug build to the release you eventually share.

Make a git commit right now, before touching anything. When a later change breaks the build in a confusing way, a diff against the known-working template answers questions faster than guessing.

One mistake I see often: picking the **Multiplatform** template because it sounds more capable. It adds iOS assumptions you do not need, and the extra conditional compilation gets in the way while

you learn. Choose the plain macOS App template. You can add more destinations later from the target settings if you ever want them.

Read the App and scene structure

Understand how the SwiftUI App entry point creates scenes and how WindowGroup supplies the application's primary windows.

Open the app's entry point file. Everything the app does starts here, so it is worth reading slowly.

```
@main
struct NotesApp: App {
  var body: some Scene {
    WindowGroup {
      ContentView()
    }
  }
}
```

`@main` tells Swift this type is the program's entry point. The struct conforms to the `App` protocol, which requires one thing: a `body` that returns a **scene**.

A **scene** is a piece of your app's interface that macOS manages for you. You declare what it contains, and the system decides how to present it, restore it, and tear it down. This split is the core idea of SwiftUI app structure: you describe, macOS manages.

`WindowGroup` is the most common scene. It does not describe a single window. It describes a *family* of windows that all share the same root view. On macOS this has a visible consequence: run the app and press Cmd+N, or choose File → New Window. You get a second, independent window showing another `ContentView`. That is `WindowGroup` doing its job.

This is a real difference from iOS. On the iPhone your app is one full-screen scene. On the Mac, users expect to open three windows of your app side by side, and `WindowGroup` gives you that behavior for free — as long as your state can handle it. Each window gets its own copy of view-local `@State`.

You can shape the windows the scene creates. This gives new windows a sensible starting size while keeping them resizable:

```
WindowGroup {
  ContentView()
}
.defaultSize(width: 800, height: 500)
```

Keep startup work out of `body`. SwiftUI can evaluate `body` more than once, so anything with side effects — opening files, starting timers, spawning processes — does not belong there. Create services explicitly, store them as properties on the app struct, and inject them into the views that need them.

The mistake to watch for: doing setup inside `body` and assuming it runs exactly once. It does not. If you see duplicated log lines or double network requests at launch, this assumption is usually the cause. Move the work into a model object you create once and pass down.

Separate model, state, and views

Give durable data, interface state, and rendered views clear ownership before the project grows.

Before the app grows, decide who owns what. Every messy SwiftUI codebase I have seen got messy the same way: views quietly accumulated data storage, file access, and business rules until nobody could test anything.

Three kinds of things live in a SwiftUI app. **Model data** is what the app is about — the notes. **Interface state** is what the UI is doing right now — which note is selected, whether a sheet is open. **Views** render the first two and send actions back.

Start with a value type for the model:

```
struct Note: Identifiable, Codable {
  let id: UUID
  var title: String
  var body: String
}
```

A struct is the right default here. It is `Identifiable` so lists can track rows, and `Codable` so we can save it to disk later without extra work.

The collection of notes needs a single owner that views can observe:

```
@MainActor
final class NotesModel: ObservableObject {
  @Published var notes: [Note] = []

  func createNote() {
    notes.append(Note(id: UUID(), title: "New note", body: ""))
  }
}
```

Create one instance at the app level and inject it:

```
@main
struct NotesApp: App {
  @StateObject private var model = NotesModel()

  var body: some Scene {
    WindowGroup {
      ContentView()
        .environmentObject(model)
    }
  }
}
```

`@StateObject` makes the app own the model's lifetime. Views read it with `@EnvironmentObject` and call methods like `createNote()` instead of mutating arrays directly. The method names become the vocabulary of what your app can do.

Interface state stays local. The selected note ID or a "show settings sheet" flag belongs in `@State` inside the view that uses it. If you push every ephemeral flag into the shared model, every window of your app will fight over the same selection.

The payoff shows up in testing. `NotesModel` is a plain class. You can create one in a unit test, call `createNote()`, and assert on `notes` — no window, no simulator, no waiting. If your important behavior can only be tested by clicking through the app, the ownership boundaries are in the wrong place.

Preview and run real windows

Use previews for focused visual work, then test the same view in resizable macOS windows and both appearance modes.

Xcode previews render one view without launching the app. For layout work they cut the feedback loop from a full launch to about a second.

```
#Preview("Empty") {
  ContentView()
    .environmentObject(NotesModel())
    .frame(width: 700, height: 450)
}
```

The `frame` matters on macOS. Without it the preview picks an arbitrary size, and you end up judging a Mac window layout in a phone-shaped canvas.

Add a second preview for realistic content. The trick is a model preloaded with data:

```
#Preview("With notes") {
  let model = NotesModel()
  model.notes = [
    Note(id: UUID(), title: "Groceries", body: "Milk, eggs, coffee"),
    Note(id: UUID(), title: "Ideas", body: "A long body to check how text wraps in the detail
  ]
  return ContentView()
    .environmentObject(model)
    .frame(width: 700, height: 450)
}
```

Build a small library of these: empty, populated, one note with a very long title, and a dark mode variant with `.preferredColorScheme(.dark)`. Each preview encodes an assumption about your layout. When one breaks, you find out immediately instead of during a demo.

Then run the actual app, because previews lie by omission. A preview never opens a real window. It does not run your menu commands, does not restore window frames, does not ask for file permissions, and does not load persisted state. Cmd+R exercises all of that.

While the app is running, do a short manual pass. Resize the window down to its smallest useful size and watch what truncates. Open a second window with Cmd+N and change a note — both windows should update, because they share one model. Tab through the controls to check keyboard focus.

The mistake here is treating a green preview as proof the app works. I have seen a layout look perfect in previews and clip its toolbar in a real window, because the preview frame was taller than the window's restored size. Previews are one tool for one view. The running app is the truth.

Check your understanding: starting a macOS app

Check the key ideas from starting a macos app before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Windows and commands

Build a sidebar, application menus, keyboard shortcuts, utility windows, and settings.

Build a sidebar interface

Use NavigationSplitView to create a Mac-friendly sidebar, selection, and detail layout driven by stable identifiers.

Look at Mail, Notes, or Finder. The Mac pattern is the same everywhere: a sidebar list on the left, the selected item's detail on the right. `NavigationSplitView` gives you that structure with the platform behavior users expect — a draggable divider, a toolbar button to collapse the sidebar, correct resizing.

On iOS you might reach for `NavigationStack` and push views. On the Mac, navigation is mostly *selection*: nothing gets pushed, the detail area shows whatever is selected. `NavigationSplitView` models exactly that.

Drive it with a selection binding that stores an identifier, not a whole note:

```
struct ContentView: View {
  @EnvironmentObject private var model: NotesModel
  @State private var selection: Note.ID?

  var body: some View {
    NavigationSplitView {
      List(model.notes, selection: $selection) { note in
        Text(note.title)
      }
      .navigationSplitViewColumnWidth(min: 180, ideal: 220)
    } detail: {
      NoteDetail(id: selection)
    }
  }
}
```

Because `Note` is `Identifiable`, the list tags each row with the note's ID automatically. The detail view receives that ID and asks the model for the current data:

```
struct NoteDetail: View {
  @EnvironmentObject private var model: NotesModel
  let id: Note.ID?

  var body: some View {
    if let note = model.notes.first(where: { $0.id == id }) {
```

```
      Text(note.body)
        .frame(maxWidth: .infinity, maxHeight: .infinity)
    } else {
      Text("Select a note")
        .foregroundStyle(.secondary)
    }
  }
}
```

Why an ID and not the note itself? The model stays the single source of truth. If the selection held a copy of the note, editing it elsewhere would leave the detail showing stale data. With an ID, the detail always resolves fresh state, and when the note is deleted the lookup fails cleanly instead of showing a ghost.

Run the app and check the behavior you got for free. Drag the divider. Click the sidebar toggle in the toolbar. Use the arrow keys in the list — selection follows the keyboard, which Mac users absolutely expect.

Then test the empty cases on purpose. Launch with no notes: the detail should show the placeholder, not crash. Select a note and delete it: `selection` now points at a note that no longer exists, the `first(where:)` lookup returns `nil`, and the placeholder comes back. Absence is a normal state for selection. Code that assumes "something is always selected" is the most common crash in this layout.

Add menu commands and shortcuts

Expose important actions through normal macOS menus and give frequent actions discoverable keyboard shortcuts.

A Mac app without menu bar items feels broken. Users look for actions in the menus, search for them with the Help menu, and learn shortcuts from the labels shown next to each item. On iOS none of this exists — the menu bar is the most Mac-specific surface you will build.

SwiftUI attaches menus to scenes with the `.commands` modifier. Add it to the `WindowGroup`:

```
WindowGroup {
  ContentView()
    .environmentObject(model)
}
.commands {
  CommandGroup(after: .newItem) {
    Button("New Note") {
      model.createNote()
    }
    .keyboardShortcut("n", modifiers: [.command, .shift])
  }
}
```

`CommandGroup(after: .newItem)` inserts your button into the File menu, right after the standard New item. SwiftUI turns the `Button` into a real menu item and the `keyboardShortcut` into ⇧⌘N, displayed beside the label.

When your actions deserve their own top-level menu, use `CommandMenu`:

```
CommandMenu("Notes") {
  Button("Sort by Title") { model.sortByTitle() }

  Divider()

  Button("Delete All Notes", role: .destructive) {
    model.deleteAll()
  }
  .disabled(model.notes.isEmpty)
}
```

Commands placed at the scene level work no matter which view has focus. That is the point: put them on the scene, not buried inside a view, and they stay available while the user moves between windows.

Disable commands that make no sense right now. The `.disabled(model.notes.isEmpty)` line keeps "Delete All Notes" grayed out when there is nothing to delete. A grayed-out item communicates state. An item that clicks and silently does nothing communicates that your app is buggy.

Run the app and verify. The File menu should show New Note with N beside it. Press the shortcut — a note appears. Then open the Help menu and type "new" into the search field: macOS finds your command and points a floating arrow at it. You wrote none of that.

The classic mistake is claiming a shortcut the system already uses. N, for example, already means New Window for a `WindowGroup` scene. Assign it to New Note and you silently take a standard feature away from your users. Check the existing menus before choosing a shortcut, and keep the plain -letter combinations for the most standard actions.

Open a purpose-built window

Declare a window with a stable scene identifier and open it from an action without manually managing NSWindow instances.

Not every window shows the same content. Our notes app might want an activity log — one utility window, opened on demand, never duplicated. On the Mac this is normal app behavior. On iOS it barely exists as a concept.

Before SwiftUI, this meant creating and retaining an `NSWindow` yourself, positioning it, and remembering not to create a second one. The `Window` scene replaces all of that with a declaration:

```
@main
struct NotesApp: App {
  @StateObject private var model = NotesModel()

  var body: some Scene {
    WindowGroup {
      ContentView()
        .environmentObject(model)
    }

    Window("Activity", id: "activity") {
      ActivityView()
        .environmentObject(model)
```

```
    }
  }
}
```

`Window` is the single-instance sibling of `WindowGroup`. Where `WindowGroup` happily creates as many windows as the user asks for, `Window` guarantees at most one. The `id` is how the rest of the app refers to it.

To open it from a view, grab the `openWindow` action from the environment:

```
struct ContentView: View {
  @Environment(\.openWindow) private var openWindow

  var body: some View {
    Button("Show Activity") {
      openWindow(id: "activity")
    }
  }
}
```

You can trigger the same action from a menu command, which is where it usually belongs. Environment actions are not available on the `App` struct itself, so wrap the button in a small view:

```
.commands {
  CommandGroup(after: .windowArrangement) {
    OpenActivityCommand()
  }
}

struct OpenActivityCommand: View {
  @Environment(\.openWindow) private var openWindow

  var body: some View {
    Button("Show Activity") {
      openWindow(id: "activity")
    }
    .keyboardShortcut("1", modifiers: [.command, .option])
  }
}
```

Run it and click the button twice. The first click opens the window. The second brings the existing one to the front instead of spawning a duplicate. That de-duplication is the whole reason the scene has a stable identifier — SwiftUI uses the `id` to find the scene, and `Window` uses it to enforce "only one".

Check the Window menu too: your Activity window is listed there automatically, and macOS restores it on relaunch if it was open.

The mistake to avoid is declaring a `WindowGroup` for something that should be a `Window`. Everything appears to work, until a user triggers the open action twice and ends up with two activity logs drifting out of sync. If duplicates of a window would ever confuse the user, it is a `Window`, not a `WindowGroup`.

Add a Settings scene

Store small preferences with AppStorage and let SwiftUI connect a normal settings window to the application menu.

Every Mac app has a Settings item in its application menu, with , as the shortcut. Users do not think about this — they expect it, the way they expect Quit at the bottom of the same menu.

SwiftUI wires all of it from one scene declaration. Add it beside your other scenes:

```
Settings {
  SettingsView()
}
```

That is the whole integration. The menu item appears, , works, and macOS presents a standard settings window. You never write "open the settings window" code.

Inside, build a form. For small preferences, **@AppStorage** connects a property straight to UserDefaults:

```
struct SettingsView: View {
  @AppStorage("showLineNumbers") private var showLineNumbers = true
  @AppStorage("fontSize") private var fontSize = 14.0

  var body: some View {
    Form {
      Toggle("Show line numbers", isOn: $showLineNumbers)
      Slider(value: $fontSize, in: 10...24) {
        Text("Font size")
      }
    }
    .padding()
    .frame(width: 350)
  }
}
```

Reading the same key elsewhere gives you a live value. Declare `@AppStorage("showLineNumbers")` in the note detail view and the UI updates the moment the toggle changes — no notification code, no manual refresh.

Two rules keep this clean. First, key names are permanent: rename `"showLineNumbers"` in a later version and every user silently loses their preference. Define keys once and never touch them again. Second, every key needs a sensible default, because the first launch has no stored value.

If settings grow, split them into tabs the way system apps do:

```
TabView {
  GeneralSettings()
    .tabItem { Label("General", systemImage: "gearshape") }
  EditorSettings()
    .tabItem { Label("Editor", systemImage: "textformat") }
}
```

Know what does not belong here. UserDefaults stores plain text in a plist anyone can read with one Terminal command, so API tokens and passwords go in the Keychain — we will do exactly that in

a later lesson. Primary documents, like the notes themselves, need real files with atomic saves, not preference storage.

Verify by running the app: the application menu shows Settings…, , opens the window, toggling the switch updates the editor immediately, and the value survives a relaunch. If the menu item is missing, your `Settings` scene is probably declared inside another scene's body instead of beside it.

Check your understanding: windows and commands

Check the key ideas from windows and commands before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Files and persistence

Choose storage locations, save atomically, accept user files, and handle replacement.

Choose the right storage location

Put application-owned data in Application Support and user-created documents where the user chooses them.

Your app has data to save. The first question is not *how* — it is *where*. macOS has conventions about where files live, and apps that ignore them lose user data at update time or fill folders that backups skip.

One place is off limits: the app bundle. The `.app` you ship is signed, and the signature covers every byte inside it. Writing into your own bundle breaks the signature, and the next update replaces the bundle wholesale, taking your "saved" data with it. Treat the bundle as read-only, always.

Data your app owns and manages — the notes database, in our case — belongs in **Application Support**. Ask `FileManager` for it, then add a folder named for your app:

```
let base = FileManager.default.urls(
  for: .applicationSupportDirectory,
  in: .userDomainMask
)[0]
let folder = base.appending(path: "Notes")

try FileManager.default.createDirectory(
  at: folder,
  withIntermediateDirectories: true
)
```

Always ask the API instead of hardcoding `~/Library/Application Support`. In a sandboxed app — and new Xcode projects are sandboxed by default — the real location is inside your app's container, something like `~/Library/Containers/com.flaviocopes.Notes/Data/Library/Application Support/Notes`. The API resolves that for you.

`createDirectory` with `withIntermediateDirectories: true` is safe to call on every launch. It creates the folder the first time and quietly succeeds when it already exists.

Redownloadable or regenerable data goes in **Caches** instead:

```
let caches = FileManager.default.urls(
  for: .cachesDirectory,
  in: .userDomainMask
)[0]
```

The system may delete caches under disk pressure, and backups do not guarantee them. That is exactly right for thumbnails or downloaded previews, and exactly wrong for the only copy of the user's notes.

Documents the *user* owns — an exported notes archive, say — are different again. Do not bury them in Application Support where nobody will find them. Let the user pick the location with a save panel, which we cover two lessons from now.

Verify your choice by running the app, saving, and finding the file in Finder. Go to Folder (G) with the container path gets you there. Then think through uninstall: dragging your app to the Trash leaves Application Support data behind, which is normal on macOS, but worth documenting for your users.

The mistake to recognize: building paths with string concatenation and `NSHomeDirectory()`. It works in development, then breaks the moment sandboxing changes the layout. `FileManager.urls(for:in:)` is the contract. String paths are a guess.

Save Codable data atomically

Encode one small model, write it atomically, and replace the previous file only after the new bytes are ready.

Our notes need to survive a relaunch. For a small local app you do not need a database — a single JSON file, written carefully, carries you a long way.

Since `Note` already conforms to `Codable`, encoding the whole array is two lines:

```
func save(_ notes: [Note], to fileURL: URL) throws {
  let data = try JSONEncoder().encode(notes)
  try data.write(to: fileURL, options: .atomic)
}
```

The `.atomic` option is the part people skip, and it is the part that matters. Without it, `write` streams bytes straight into the destination file. If the app crashes or the Mac loses power halfway through, the file is left half-written — the old data is gone and the new data is garbage.

An **atomic write** goes through a temporary file. The full new content is written beside the destination, and only then renamed into place. The rename is a single filesystem operation: at every moment the path holds either the complete old file or the complete new one, never a mix.

Loading needs more care than saving, because "no file" and "broken file" are different situations:

```
func load(from fileURL: URL) throws -> [Note] {
  let data: Data
  do {
    data = try Data(contentsOf: fileURL)
  } catch CocoaError.fileReadNoSuchFile {
    return []
```

```
  }
  return try JSONDecoder().decode([Note].self, from: data)
}
```

A missing file is normal — it is the first launch, and an empty list is the right answer. A file that exists but fails to decode is *not* normal. Let that error propagate and show the user something, because it means their data is there but unreadable.

Here is the mistake that destroys data: wrapping the whole load in `try?` and falling back to `[]`. Decoding fails for some reason, the app starts "fresh" with zero notes, and the next autosave writes that empty array over the user's file. The corrupt-but-recoverable data is now actually gone. Distinguish the errors, and never let a failed read feed the next write.

When you later change the `Note` structure, copy the old file to a backup name before migrating. A migration that fails halfway with no backup is the same story with extra steps.

Verify with a crash test. Save a few notes, find the JSON in Application Support, and open it — it should be readable and complete. Then force-quit the app mid-use a few times. The file should always parse. If you ever find half a JSON document in there, an unatomic write slipped in somewhere.

Let the user choose a file

Use fileImporter for user-selected files and keep sandbox access scoped to the URL the user approved.

Sandboxed Mac apps cannot read arbitrary files, and that is by design. Your app sees its own container and nothing else. The way to reach the user's files is **user intent**: the user picks a file in a system dialog, and that choice *is* the permission grant.

This is the piece that surprises people coming from other platforms. There is no "files permission" to request up front. Access arrives one URL at a time, attached to an explicit user action.

SwiftUI exposes the open panel through the `fileImporter` modifier. Suppose we let users import a text file into their notes:

```
struct ImportButton: View {
  @EnvironmentObject private var model: NotesModel
  @State private var showImporter = false

  var body: some View {
    Button("Import Text File…") {
      showImporter = true
    }
    .fileImporter(
      isPresented: $showImporter,
      allowedContentTypes: [.plainText]
    ) { result in
      if let url = try? result.get() {
        model.importNote(from: url)
      }
    }
  }
}
```

The panel that appears is the real system open panel, running outside your process. Your app never sees the filesystem during browsing — it receives only the final, approved URL.

That URL is **security-scoped**. Before reading it you must start access, and you must stop when done:

```
func importNote(from url: URL) {
  guard url.startAccessingSecurityScopedResource() else { return }
  defer { url.stopAccessingSecurityScopedResource() }

  if let text = try? String(contentsOf: url, encoding: .utf8) {
    notes.append(Note(id: UUID(), title: url.lastPathComponent, body: text))
  }
}
```

`defer` guarantees the stop call runs however the function exits. Keep the window between start and stop short — do the read, then get out.

The grant does not survive a relaunch. If your app needs to reopen the same file tomorrow, store a **security-scoped bookmark**:

```
let bookmark = try url.bookmarkData(
  options: .withSecurityScope
)
// persist the Data, then later:
var stale = false
let restored = try URL(
  resolvingBookmarkData: bookmark,
  options: .withSecurityScope,
  bookmarkDataIsStale: &stale
)
```

The bookmark encodes the permission itself, so resolving it after relaunch restores access without asking the user again.

The classic mistake: saving `url.path` as a string and trying to read it on the next launch. The path is correct, the file exists, and the read fails with a permission error anyway, because the string carried none of the grant. If a file works until the app restarts, this is almost always what happened.

Verify both halves: import a file from your Desktop and see the note appear, then relaunch and confirm the bookmark-restored URL still reads.

Handle file replacement

Treat replacement as a different event from appending bytes and reopen watched files when their identity changes.

Suppose the notes app watches an external file — a log the user imported, re-rendered whenever it changes. File watching on macOS has a trap that catches nearly everyone: the file you watch can stop being the file at that path.

Remember the atomic save from two lessons ago? Editors do the same thing. When TextEdit or VS Code saves, it writes a new file and renames it over the old path. The path is unchanged, but the file — the actual inode your watcher holds open — is now an orphan. Your watcher keeps listening

to a file nothing will ever write to again.

The low-level tool is a dispatch source attached to an open file descriptor:

```
let descriptor = open(url.path, O_EVTONLY)
let watcher = DispatchSource.makeFileSystemObjectSource(
  fileDescriptor: descriptor,
  eventMask: [.write, .rename, .delete],
  queue: .main
)
watcher.setEventHandler {
  handle(watcher.data)
}
watcher.resume()
```

The event mask is the key decision. `.write` alone covers direct appends. `.rename` and `.delete` are how atomic replacement shows up — the old file gets renamed or removed, and that event is your signal that the descriptor is dead.

When one of those events arrives, do not keep reading. Tear down and reattach:

```
func handle(_ event: DispatchSource.FileSystemEvent) {
  if event.contains(.rename) || event.contains(.delete) {
    watcher.cancel()
    close(descriptor)
    // the path may briefly not exist during the swap
    DispatchQueue.main.asyncAfter(deadline: .now() + 0.1) {
      startWatching(url)
    }
  } else {
    readNewData()
  }
}
```

After reopening, be careful with your read position. If you were tailing the file from a saved offset, the replacement file may be shorter than that offset. Check the new size first, and treat "smaller than before" as truncation: start from zero instead of reading from a position past the end.

Debounce, too. Editors often produce a burst of events per save, and collapsing a burst into one reload is good. What you must not do is collapse a rename into a plain write, because they require different responses.

Test both paths deliberately. Running `echo hello >> watched.txt` in Terminal produces a pure append — your `.write` handler should fire. Saving the same file from TextEdit produces the rename dance — your reattach path should fire. If your watcher survives the first test but goes silent after the second, you are watching the descriptor and ignoring identity, which is exactly the bug this lesson exists to prevent.

Check your understanding: files and persistence

Check the key ideas from files and persistence before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

macOS integration

Add a menu bar extra, notifications, Keychain storage, and explicit command-line tools.

Build a menu bar extra

Expose a small set of frequent actions from the system menu bar without turning the extra into a second complete application.

The icons at the right end of the menu bar are **menu bar extras**. They exist for one job: giving the user something quick without making them find your window. For our notes app, that means capturing a thought in two clicks from inside any other app.

SwiftUI turned what used to be a pile of `NSStatusItem` code into a scene. Declare it beside the others:

```
MenuBarExtra("Notes", systemImage: "note.text") {
  MenuBarContent()
    .environmentObject(model)
}
```

The content is a normal view, so environment actions work inside it:

```
struct MenuBarContent: View {
  @EnvironmentObject private var model: NotesModel
  @Environment(\.openWindow) private var openWindow

  var body: some View {
    Button("New Note") { model.createNote() }

    Divider()

    Button("Open Notes") { openWindow(id: "main") }
  }
}
```

For that last button to work, give the main scene an identifier: `WindowGroup(id: "main") { … }`.

By default the content renders as a **menu** — a plain list of commands, like the Wi-Fi icon. There is a second style that shows a full view in a floating panel:

```
MenuBarExtra("Notes", systemImage: "note.text") {
  QuickNoteView()
    .frame(width: 300, height: 200)
}
.menuBarExtraStyle(.window)
```

Choose the menu style unless the content genuinely needs controls a menu cannot hold, like a text field for typing a quick note. The window style is heavier and behaves differently — it stays open while the user interacts, and you become responsible for making it feel dismissable.

The string label matters even though users see the icon. VoiceOver reads it, and it identifies the

item when users rearrange extras by -dragging them.

Run the app and look at the top right of the screen. The note icon should sit among the system extras. Click it, create a note from the menu, then open the main window and confirm the note is there — both scenes share one `model`, which is why we inject the same instance everywhere.

The mistake with menu bar extras is scope creep. It starts as three commands, then grows tabs, settings, and scrolling lists, until you have built a second app inside a popover. When the extra needs that much, the answer is opening the real window. Keep the extra as the shortcut, not the destination.

One more warning: if the extra becomes the app's *only* interface, with the Dock icon hidden, add an explicit Quit button to its menu. Without a Dock icon, that menu is the only place the user can quit from.

Request notification permission in context

Ask for notification permission when the user enables a feature that needs it, then handle denial without breaking the app.

Notifications interrupt people. macOS gates them behind a permission prompt, and users have trained themselves to click "Don't Allow" on prompts they do not understand. The single biggest thing you control is *when* the question appears.

Ask at first launch, out of nowhere, and you get denied. Ask at the moment the user turns on a feature that clearly needs notifications — "Remind me about this note" — and the prompt explains itself.

The API lives in the UserNotifications framework:

```
import UserNotifications

func enableReminders() async throws -> Bool {
  let center = UNUserNotificationCenter.current()
  return try await center.requestAuthorization(
    options: [.alert, .sound]
  )
}
```

The first call shows the system prompt. Every later call returns the stored decision without any UI, so calling it from the feature toggle is safe.

Sending a notification is a separate step — content, trigger, request:

```
func scheduleReminder(for note: Note, in seconds: TimeInterval) async throws {
  let content = UNMutableNotificationContent()
  content.title = "Reminder"
  content.body = note.title

  let trigger = UNTimeIntervalNotificationTrigger(
    timeInterval: seconds,
    repeats: false
  )
  let request = UNNotificationRequest(
```

```
    identifier: note.id.uuidString,
    content: content,
    trigger: trigger
  )
  try await UNUserNotificationCenter.current().add(request)
}
```

Using the note's ID as the request identifier is deliberate: scheduling again with the same identifier replaces the old reminder instead of stacking a duplicate.

Denial is a state, not an error. The user can refuse now and change their mind in System Settings later, or the reverse. Check the current status before scheduling and keep the feature honest:

```
let settings = await UNUserNotificationCenter.current()
  .notificationSettings()
if settings.authorizationStatus == .denied {
  // show a "turn on notifications in System Settings" hint
}
```

Now the part that trips everyone during development: schedule a reminder for five seconds, keep the app in front, and nothing appears. That is not a bug. By default macOS suppresses banners while the sending app is frontmost — the assumption is the user can already see whatever you would tell them. Switch to another app and the banner arrives. If you want banners while frontmost, implement the `UNUserNotificationCenterDelegate` method `userNotificationCenter(_:willPresent:)` and return `.banner`.

Verify the full flow: toggle the reminder feature, see the permission prompt appear *at that moment*, accept, schedule, switch to Safari, and watch the banner arrive. Then deny permission in System Settings and confirm your app shows its hint instead of pretending the reminder was scheduled.

Store a secret in Keychain

Keep tokens and passwords out of UserDefaults and files by using a narrowly named Keychain item.

Say the notes app grows a sync feature and receives an API token. Where does it go? Not UserDefaults. Everything there sits in a plain plist on disk, readable with one command:

```
defaults read com.flaviocopes.Notes
```

Any process running as the user can do that. The **Keychain** is the answer macOS provides: an encrypted store, unlocked with the user's login, with access controlled per app.

The Keychain API is the Security framework — C functions driven by dictionaries. Do not spread these calls around your codebase. Wrap them once behind a small interface:

```
protocol SecretStore {
  func save(_ value: Data, account: String) throws
  func load(account: String) throws -> Data?
  func delete(account: String) throws
}
```

Here is the save, storing a **generic password** item identified by a service and an account:

```
func save(_ value: Data, account: String) throws {
  let query: [String: Any] = [
```

```
    kSecClass as String: kSecClassGenericPassword,
    kSecAttrService as String: "com.flaviocopes.Notes",
    kSecAttrAccount as String: account,
    kSecValueData as String: value,
  ]
  SecItemDelete(query as CFDictionary)
  let status = SecItemAdd(query as CFDictionary, nil)
  guard status == errSecSuccess else {
    throw KeychainError.unexpectedStatus(status)
  }
}
```

The service and account names are the item's identity. Name them precisely — your bundle identifier for the service, `"sync-token"` for the account — so a Keychain search never matches more than you intended.

Loading mirrors it:

```
func load(account: String) throws -> Data? {
  let query: [String: Any] = [
    kSecClass as String: kSecClassGenericPassword,
    kSecAttrService as String: "com.flaviocopes.Notes",
    kSecAttrAccount as String: account,
    kSecReturnData as String: true,
  ]
  var result: AnyObject?
  let status = SecItemCopyMatching(query as CFDictionary, &result)
  if status == errSecItemNotFound { return nil }
  guard status == errSecSuccess else {
    throw KeychainError.unexpectedStatus(status)
  }
  return result as? Data
}
```

`errSecItemNotFound` is a normal answer — the user has not signed in yet — so it becomes `nil` rather than a thrown error. Every other non-success status is worth surfacing, mapped to an error type. Never print the secret itself while debugging, not even temporarily. Logs outlive debugging sessions.

The protocol earns its keep in tests. Give the test target an in-memory implementation backed by a dictionary, and your sign-in logic can be tested for success, missing token, and Keychain failure without touching the real Keychain.

Verify with a round trip: save a token, quit, relaunch, load it back. You can also inspect the item in the Keychain Access app — search for your service name. And remember the boundary: the Keychain protects the token at rest. Once your code loads it, where it travels — logs, error reports, URLs — is entirely your responsibility.

Launch a command-line tool explicitly

Resolve an executable deliberately and give Process an explicit environment instead of relying on a Finder-launched app's PATH.

A Mac app can lean on command-line tools for heavy lifting — imagine the notes app shelling out to `git` to version a notes folder. The API is `Process`, and the first thing to learn about it is that your app does not live in your shell.

When you type `git` in Terminal, the shell searches your `PATH`, shaped by `.zshrc` and Homebrew's setup. An app launched from Finder inherits none of that. Its environment is minimal, its `PATH` is a short system default, and a tool that works in Terminal can be unfindable from the app.

So be explicit about everything:

```
let process = Process()
process.executableURL = URL(filePath: "/usr/bin/git")
process.arguments = ["--version"]
process.environment = [
  "PATH": "/opt/homebrew/bin:/usr/local/bin:/usr/bin:/bin"
]
```

The absolute `executableURL` removes the `PATH` question for system tools. For user-installed tools the honest options are checking the known locations — `/opt/homebrew/bin` on Apple silicon, `/usr/local/bin` on Intel — or letting the user pick the executable with the file importer from the previous module.

To read output, attach a pipe before launching:

```
let stdout = Pipe()
process.standardOutput = stdout

try process.run()

let data = try stdout.fileHandleForReading.readToEnd() ?? Data()
process.waitUntilExit()

let output = String(decoding: data, as: UTF8.self)
let succeeded = process.terminationStatus == 0
```

Order matters here more than it looks. Read the pipe *before* `waitUntilExit`, not after. A pipe buffer holds around 64 KB. If the child prints more while nobody reads, it blocks on its own write call, and your app blocks in `waitUntilExit` waiting for a child that is waiting for you. Both sides sit there forever. This deadlock is the most common `Process` bug, and it only appears when output grows past the buffer — which means it appears in production, not in your quick test.

Check `terminationStatus` every time. Zero means success by convention. Anything else means the tool failed, and stderr usually says why — capture it with a second pipe.

A child process is an external dependency. It can be missing, be the wrong version, hang, or print unexpected output. Record the resolved path and version at startup — `git --version` is cheap — and impose a timeout so a hung child cannot hang your app with it.

Verify from the right context: not from Xcode, which launches your app with its own environment, but from Finder. Build the app, double-click the `.app`, and run the feature. If your tool resolution is wrong, this is where it shows.

Check your understanding: macOS integration

Check the key ideas from macos integration before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Make the app reliable

Define process protocols, stop work safely, cross concurrency boundaries, and finish a tested utility.

Treat child output as a protocol

Define stable machine-readable output instead of scraping prose meant for a person.

The previous module launched tools and read their output in one gulp. That breaks down the moment you need *progress* — a long export where the interface shows a moving bar. Now the child's stdout is not text you glance at. It is an interface between two programs, and it deserves the same care as a network API.

Scraping human-oriented output is the tempting shortcut. The tool prints `Processed 12 of 40 files`, so you match it with a regex and ship. Then a later version of the tool rewords its messages, your regex silently stops matching, and the progress bar freezes at zero while the work completes fine. Nobody wrote a bug. A sentence changed.

The fix is to define the messages. If you control the tool, make it emit **JSON Lines** — one JSON object per line — on stdout, and keep human diagnostics on stderr:

```
{"event":"progress","completed":12,"total":40}
{"event":"finished","output":"recording.mp4"}
```

Each line stands alone. Decode complete lines into a typed value so unknown input fails loudly at the boundary:

```
struct ToolEvent: Decodable {
  let event: String
  let completed: Int?
  let total: Int?
  let output: String?
}

func parse(line: Data) throws -> ToolEvent {
  try JSONDecoder().decode(ToolEvent.self, from: line)
}
```

Streaming reads deliver arbitrary chunks, not lines. A read can end mid-message, so buffer until a newline appears:

```
var buffer = Data()

func receive(_ chunk: Data) {
  buffer.append(chunk)
  while let newline = buffer.firstIndex(of: UInt8(ascii: "\n")) {
```

```
    let line = buffer[..<newline]
    buffer.removeSubrange(...newline)
    if let event = try? parse(line: line) {
      handle(event)
    }
  }
}
```

Decoding failures at this boundary are information. Keep a bounded tail of recent raw lines — the last hundred, say — so when the child fails you can show or log what it actually said, instead of a bare "exit code 1".

If the protocol will evolve, add a version field to the first message and reject major versions you do not understand. That turns a future incompatibility into a clear error instead of a subtle misparse.

Verify with hostile input. Feed the parser a message split across two chunks, two messages in one chunk, and a line of plain prose. The first two must decode correctly. The third must be rejected without taking the app down. If your parser only handles one-message-per-read, it works in tests and fails under real pipe timing.

Stop a child process gracefully

Request a clean stop, wait for bounded cleanup, and escalate only when the child does not finish.

Starting a child process is easy. Stopping one is where apps corrupt files. Imagine the child is exporting a video, or compacting the notes database — kill it mid-write and the output is trash.

There is a hierarchy of stops. `interrupt()` sends SIGINT, the same signal as Ctrl+C, and a well-behaved tool catches it and finalizes its output. `terminate()` sends SIGTERM, a firmer request. SIGKILL cannot be caught at all — the process stops mid-instruction, files half-written.

Model the shutdown as a sequence with a deadline:

```
func stop(_ process: Process, deadline: TimeInterval = 5) async {
  process.interrupt()

  let exited = await waitForExit(process, timeout: deadline)
  if !exited {
    process.terminate()
    _ = await waitForExit(process, timeout: 2)
  }
}
```

First the polite request, then a bounded wait, then escalation. Each step is observable: your model can publish `stopping`, `waiting`, and `forceStopping` states, and the interface can show what is happening instead of freezing on a spinner.

Pick the deadline from what the child actually does. A tool finalizing an MP4 container may legitimately need a few seconds. Five is a reasonable default; forever is not. And when you do escalate, record it — a forced stop means the output may be incomplete, and the app should say so rather than present a broken file as done.

The same discipline applies when *your app* is the one being asked to stop. If the user quits while an export runs, do not vanish and orphan the child. Add an `NSApplicationDelegate` through

`@NSApplicationDelegateAdaptor` and implement:

```
func applicationShouldTerminate(
  _ sender: NSApplication
) -> NSApplication.TerminateReply {
  guard exporter.isRunning else { return .terminateNow }
  Task {
    await exporter.stopGracefully()
    NSApplication.shared.reply(toApplicationShouldTerminate: true)
  }
  return .terminateLater
}
```

`.terminateLater` pauses the quit while your cleanup runs. The contract is strict: you *must* eventually call `reply(toApplicationShouldTerminate:)`, on every path — success, failure, timeout. Miss one path and the app hangs at quit, which users experience as "it won't close" and solve with Force Quit, defeating the whole effort.

The mistake to avoid is the opposite extreme: reaching for SIGKILL first because it is reliable. It is reliable at stopping the process and equally reliable at corrupting whatever the process was writing. Escalate to it. Never start with it.

Test the full ladder. Start a long export and quit the app: it should delay briefly, then close, and the output file should be valid. Then simulate a hung child — a script that traps SIGINT and ignores it — and confirm your escalation fires and reports that cleanup was cut short.

Cross concurrency boundaries deliberately

Keep interface state on the main actor and move blocking file or process work behind asynchronous service methods.

Everything the user sees lives on the **main actor** — one protected context that owns all interface state. Blocking file reads and child-process waits do not belong there, because while the main actor is busy, the app is beachballing.

The design that works: interface state on the main actor, slow work in async services, and explicit hops between them.

Mark the model with `@MainActor`:

```
@MainActor
final class ExportModel: ObservableObject {
  @Published private(set) var status = "Ready"

  func export() async {
    status = "Exporting"
    let result = await exporter.run()
    status = result.summary
  }
}
```

Read `export()` as a story about threads. The first `status` assignment runs on the main actor. The `await` is the boundary: the model suspends, `exporter.run()` does its slow work wherever it likes, and — this is the part Swift handles for you — the method *resumes on the main actor* for the final

assignment. No dispatch calls, no queue names, and the compiler checks it.

The service side stays free of UI concerns:

```
struct ExportResult {
  let summary: String
}

struct Exporter {
  func run() async -> ExportResult {
    // launch the child process, await its exit
    ExportResult(summary: "Exported 3 notes")
  }
}
```

Services return values or throw typed errors. They never reach back into the model, never touch `@Published` properties, never "helpfully" dispatch to the main queue. The model pulls results across the boundary. The service does not push.

Here is the mistake and how it announces itself. Update `status` from a background context — a `DispatchQueue.global().async` block, a callback from an old-style API — and Xcode prints a purple runtime warning: publishing changes from background threads is not allowed. Sometimes the UI still updates and everything looks fine. Do not trust that. It is a data race that happens to be working today, and it will eventually show up as a stale label or a crash you cannot reproduce. Treat every purple warning as a bug found early. With strict concurrency checking enabled — and it is worth enabling — many of these mistakes stop compiling at all.

Cancellation deserves a decision, not an afterthought. When the user cancels an export, `Task.cancel()` only sets a flag. Your service must check `Task.isCancelled` at sensible points and decide what happens to partial output — delete the half-file, or keep it and mark it. Either is defensible, but choose.

Verify by feel: run a big export and click a menu while it runs. The window must stay responsive. If the beachball appears, something blocking snuck onto the main actor — the usual suspect is a synchronous `waitUntilExit` called outside the service.

Complete a native macOS utility

Finish a small app with a sidebar, durable data, menus, settings, one system integration, and tests for its important behavior.

You have all the pieces. The last lesson is assembly: one small utility, built end to end, that behaves like it belongs on a Mac.

Pick a scope you can finish. The notes app we have been building is one option. A command runner — saved shell tasks with captured output — is another that exercises the process material. My advice is to choose the one you would actually use, because you will test it harder.

The requirements map to what you built across the course:

- A `WindowGroup` main window that resizes sanely, plus any purpose-built `Window` the app needs
- One model owning the data, injected at the app level, with views that render state and send actions
- A `NavigationSplitView` sidebar, or a deliberate single-pane layout if the app is truly one list

- Menu commands with keyboard shortcuts, disabled when they do not apply
- A `Settings` scene with `@AppStorage` preferences
- Durable data: `Codable` models, atomic writes, Application Support

Then one — exactly one — system integration from the previous modules: a menu bar extra, imported files with bookmarks, notifications, a Keychain item, or a child process. Pick the one your app's workflow genuinely needs. Four half-wired integrations impress nobody. One that works every time is what makes the app feel finished.

Write tests where they pay. Model logic first:

```
@MainActor
func testCreateNoteAppendsToList() {
  let model = NotesModel()
  model.createNote()
  XCTAssertEqual(model.notes.count, 1)
}
```

Add a persistence round trip — save notes to a temporary directory, load them back, assert equality. Then one failure test at your integration boundary: the Keychain item is missing, the child exits nonzero, the bookmark is stale. The failure path is the one you never click manually, which is exactly why it needs a test.

Finish with a manual pass that mimics a real user rather than a developer. Build the app, quit Xcode, and launch the `.app` from Finder — this catches environment assumptions, especially around `PATH`. Resize every window to its minimum. Drive the whole workflow with the keyboard, menus and shortcuts only. Open Settings, change something, relaunch, confirm it stuck. Find your data file in Finder and check it is where you documented.

The trap in a capstone is feature accumulation — adding tags, sync, and themes while the delete command still crashes on an empty selection. Resist it. A small utility whose every path works teaches you more, and demos better, than a large one that mostly works.

When it passes all of that, you have something real: a native macOS app with correct structure, durable data, and one honest system integration. That is the foundation the next course builds on — shipping it to other people's Macs.

Check your understanding: making the app reliable

Check the key ideas from making the app reliable before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/build-macos-apps-swiftui/.