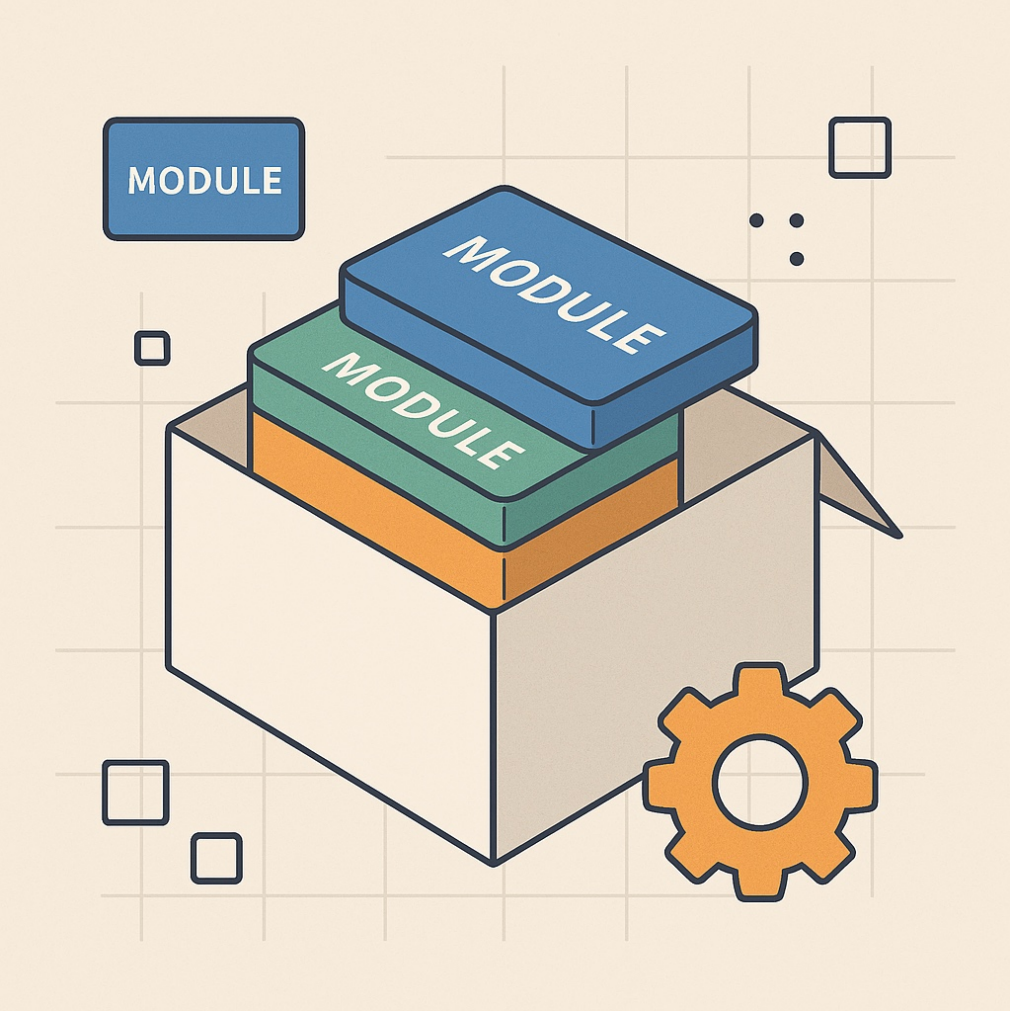
ExtremePackaging



A Modular Architecture Methodology for Swift Projects By Mihaela Mihaljevic

Introduction

Extreme Packaging is a methodology for structuring Swift projects with *maximal modularity and minimal responsibility per module.* Each module represents a single, isolated unit of logic — small enough to reason about, easy to test, and replaceable without side effects.

The core idea is **separation**: modules depend on stable interfaces, not on each other's implementations. This enables scalable architectures that remain flexible as projects evolve, while keeping build times short and dependencies explicit.

Table Of Contents:

Goals

- Promote clean boundaries between domains and features
- Enable independent development and testing per package
- Simplify refactoring and dependency management
- Keep compilation fast and the architecture transparent

Process

The repository is organized into **stages**, each representing a self-contained checkpoint in the project's evolution. When moving between stages — especially when reverting to earlier ones — always reset and clean your workspace to ensure it matches the intended state.

Here's the example for **stage 01**:

```
# Ensure you're on the branch you want
git checkout stage/01-init-packages

# Fetch the latest version
git fetch origin

# Reset your branch to the remote version
git reset --hard stage/01-init-packages

# Delete untracked files and directories
git clean -fdx
```

Philosophy

In most projects, modularization is an afterthought — introduced when the codebase becomes too large to manage. **Extreme Packaging** inverts that approach: modularization is the starting point. By treating each package as an autonomous component from day one, you gain clarity, resilience, and a foundation that naturally scales with complexity.

Part 1 — Project Setup

The first step in **Extreme Packaging** is establishing a clear and reproducible project structure that can evolve gradually through defined stages. Each stage in the repository builds upon the previous one — from the initial package setup, to workspace creation, and finally to adding the app target that ties everything together.

Overview

The project begins as a **Swift Package Manager–based structure**, designed for modularity from the start. At its core is a single `Packages` directory that houses all functional modules (such as `AppFeature`, `SharedModels`, and later others). Every addition to the project — whether a new feature, UI layer, or platform target — is layered on top of this foundation in small, trackable increments.

Here is the link to the project used: https://github.com/mihaelamj/ExtremePackaging

1.1 Stage 01 — Initialize Packages

Purpose

This stage ensures a working, self-contained Swift package that compiles and passes initial linting checks.

Parts

At **stage/01-init-packages**, the repository contains:

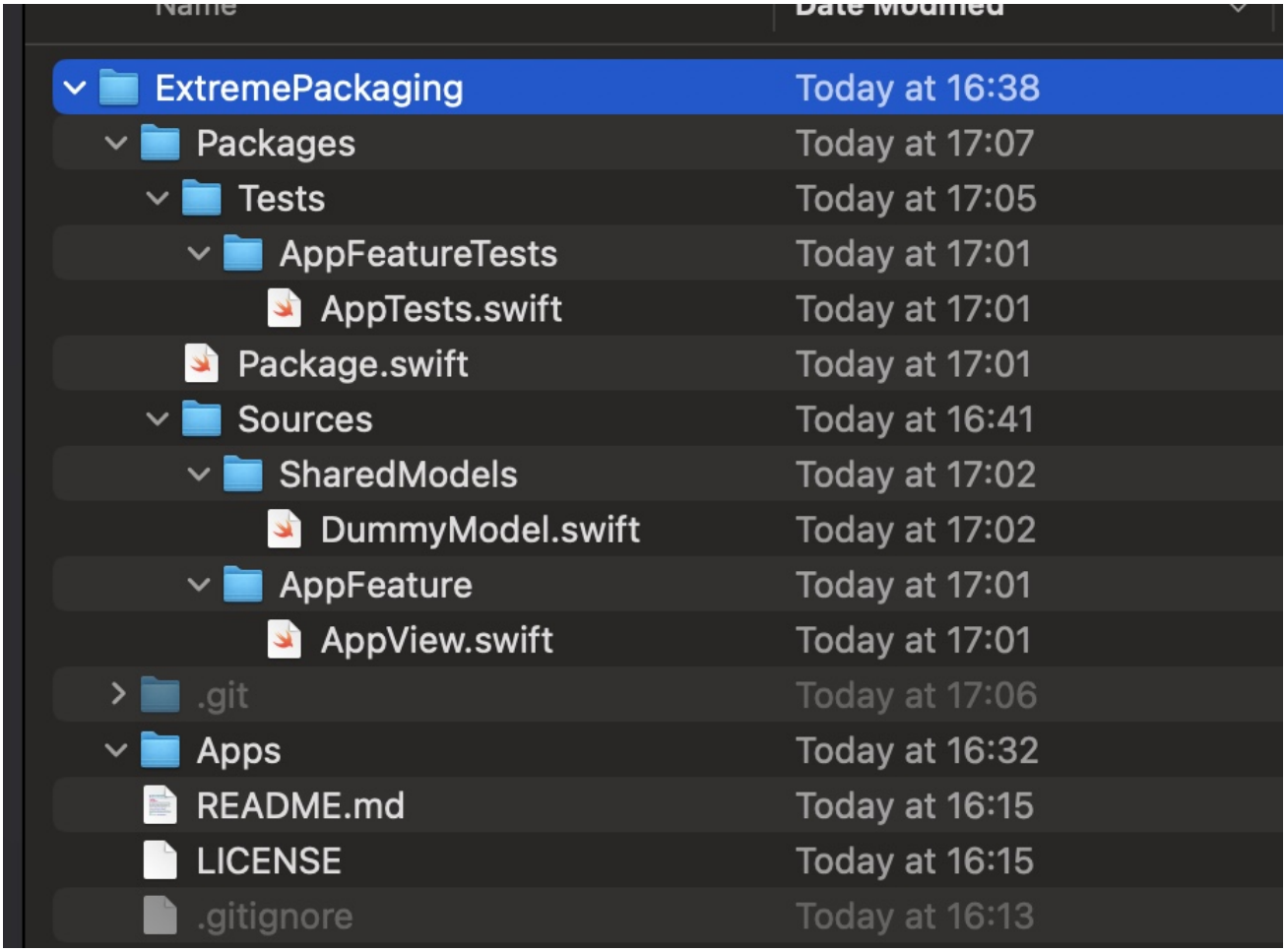
A minimal **package structure** with `Sources` and `Tests`

Core configuration files:
`.swiftlint.yml` for linting rules
`.swiftformat` for consistent formatting
`.gitignore`, `LICENSE`, and `README.md`

Two base modules:
`AppFeature` — serves as the entry feature for the app
`SharedModels` — holds simple model definitions

Folder Structure snapshot

Example folder structure:



Inside the folder:

```
?  ExtremePackaging git:(main) ? ls -all
 .
 ..
 .git
 .gitignore
 .swiftformat
 .swiftlint.yml
 Apps
 LICENSE
 Packages
 README.md
```

Inside the `Packages` folder:

```
?  ExtremePackaging git:(main) cd Packages
?  Packages git:(main) ls -all
 .
 ..
 Package.swift
 Sources
 Tests
```

Inside the `Apps` folder:

```
.
..
.gitkeep
```

Basic Package Code:

I start with package structure like this:

```
// swift-tools-version: 6.0

import PackageDescription

let package = Package(
    name: "Main",
    platforms: [
        .iOS(.v17),
        .macOS(.v14),
    ],
    products: [
        .singleTargetLibrary("AppFeature"),
    ],
    dependencies: [
        .package(url: "https://github.com/realm/SwiftLint", exact: "0.52.3"),
    ],
    targets: [
        .target(
            name: "AppFeature",
```

```
            dependencies: [
                "SharedModels",
            ]
        ),
        .testTarget(
            name: "AppFeatureTests",
            dependencies: [
                "AppFeature"
            ]
        ),
        .target(
            name: "SharedModels"
        )
    ]
)

// Inject base plugins into each target
package.targets = package.targets.map { target in
    var plugins = target.plugins ?? []
    plugins.append(.plugin(name: "SwiftLintPlugin", package: "SwiftLint"))
    target.plugins = plugins
    return target
}

extension Product {
    static func singleTargetLibrary(_ name: String) -> Product {
        .library(name: name, targets: [name])
    }
}
```

Dummy Files

AppView

```
import SharedModels
import SharedViews
import SwiftUI

public struct AppView: View {
    public var body: some View {
        VStack {
            Text("Extreme Packaging!")
                .font(.title)
                .fontWeight(.bold)
                .multilineTextAlignment(.center)
                .padding()
        }
    }
}
```

DummyModel

```
import Foundation

public struct DummyModel: Identifiable {
    public var id: UUID = .init()
    public var title: String
    public init(
        id: UUID = .init(),
        title: String
    ) {
        self.id = id
        self.title = title
    }
}
```

Stage 01-init-packages

Here's the code for **stage 01**:

```

# Clone repo if needed
git clone git@github.com:mihaelamj/ExtremePackaging.git

# Ensure you're on the branch you want
git checkout stage/01-init-packages

# Fetch the latest version
git fetch origin

# Reset your branch to the remote version
git reset --hard stage/01-init-packages

# Delete untracked files and directories
git clean -fdx
```

To see how it looks in **Xcode** we need to switch to subfolder `packages`, since we haven't created the workspace yet.

```
?  ExtremePackaging git:(stage/01-init-packages) ? cd Packages
?  Packages git:(stage/01-init-packages) ? xed .
```

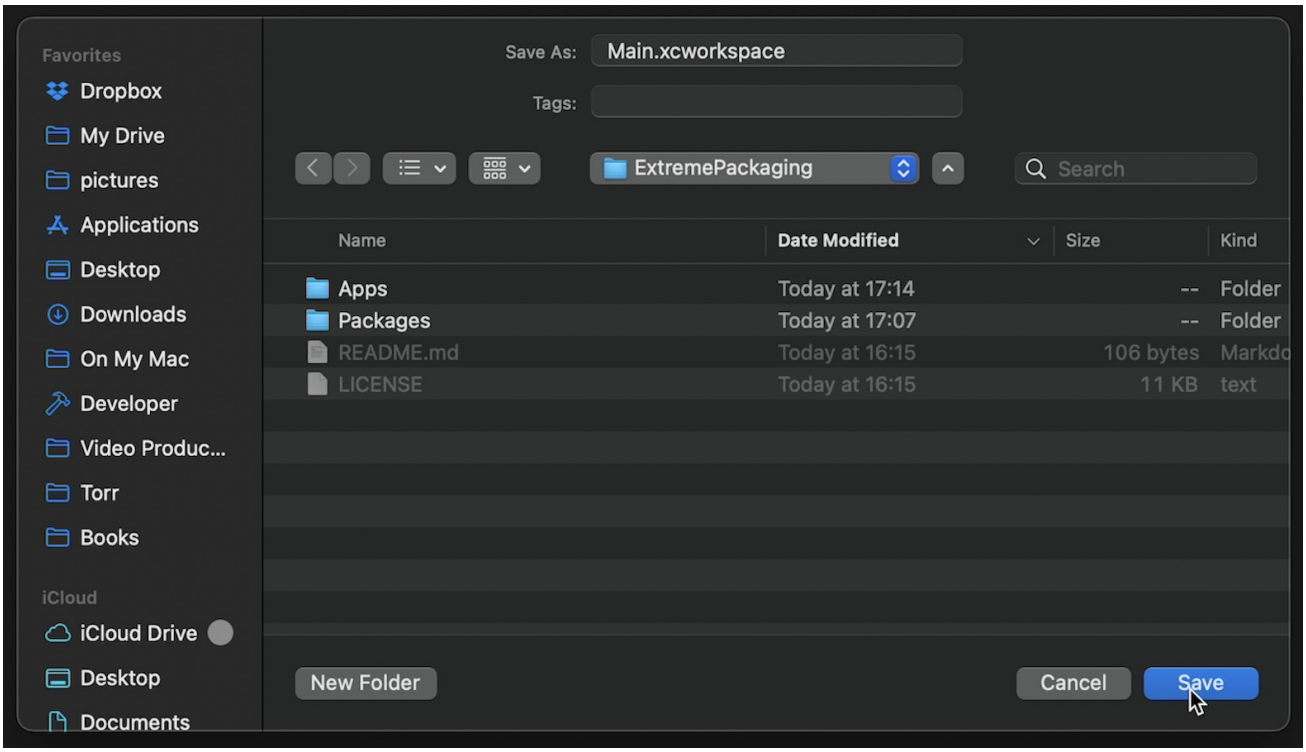
1.2 Stage 02 — Add Workspace

Purpose

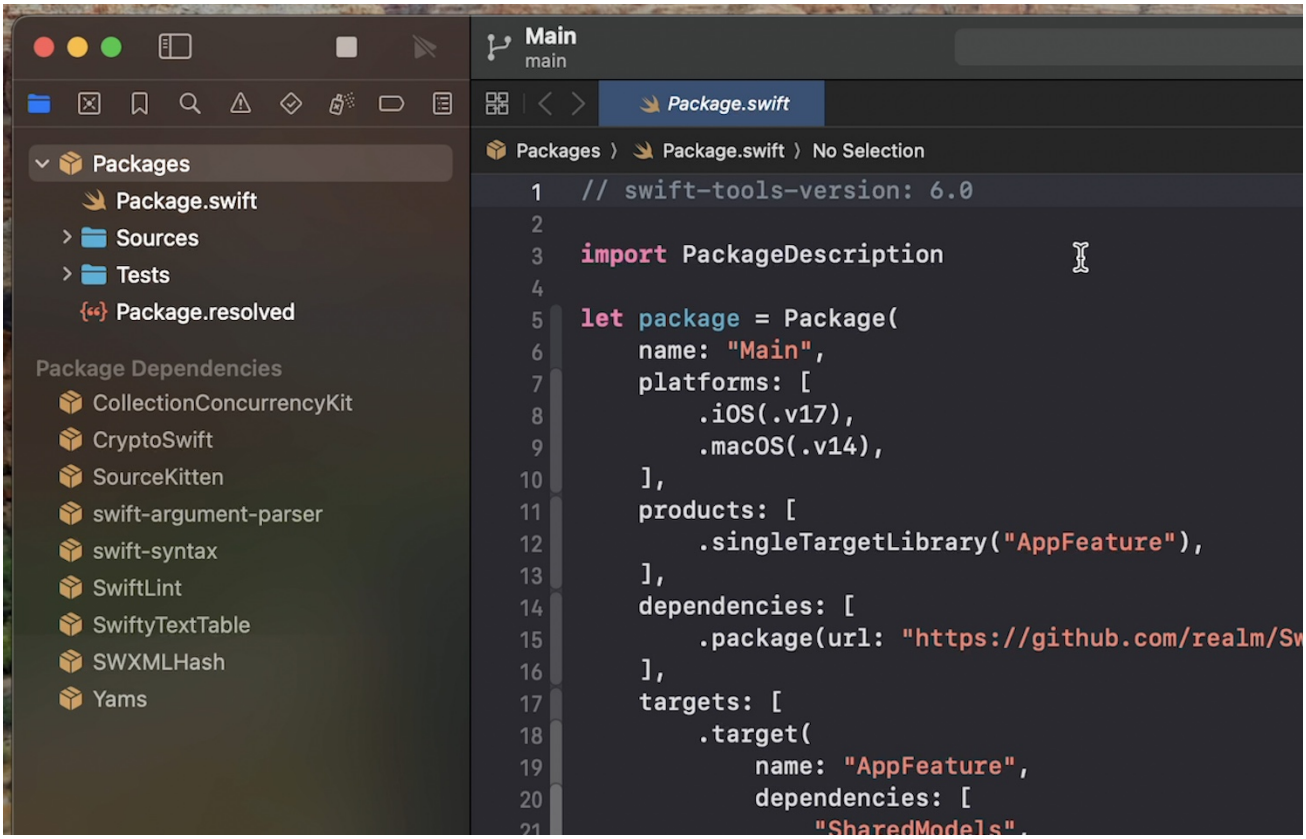
In **stage/02-workspace-added**, an Xcode **workspace** named `Main.xcworkspace` is introduced at the project root. It includes the `Packages` folder, allowing smooth integration of multiple modules while keeping them decoupled. This step establishes the foundation for a multi-target environment.

Workspace creation steps

Create a new workspace (I name it `Main`), in the root of our folder.



Add folder `Packages` to the workspace. It will add our package structure to the project:



Stage 02-add—to-workspace

Here's the code for **stage 02**:

```
# Clone repo if needed
git clone git@github.com:mihaelamj/ExtremePackaging.git

# Ensure you're on the branch you want
git checkout stage/02-add—to-workspace

# Fetch the latest version
git fetch origin

# Reset your branch to the remote version
git reset --hard stage/02-add—to-workspace

# Delete untracked files and directories
git clean -fdx
```

Now we have the **workspace**, so we just need to open **Xcode**:

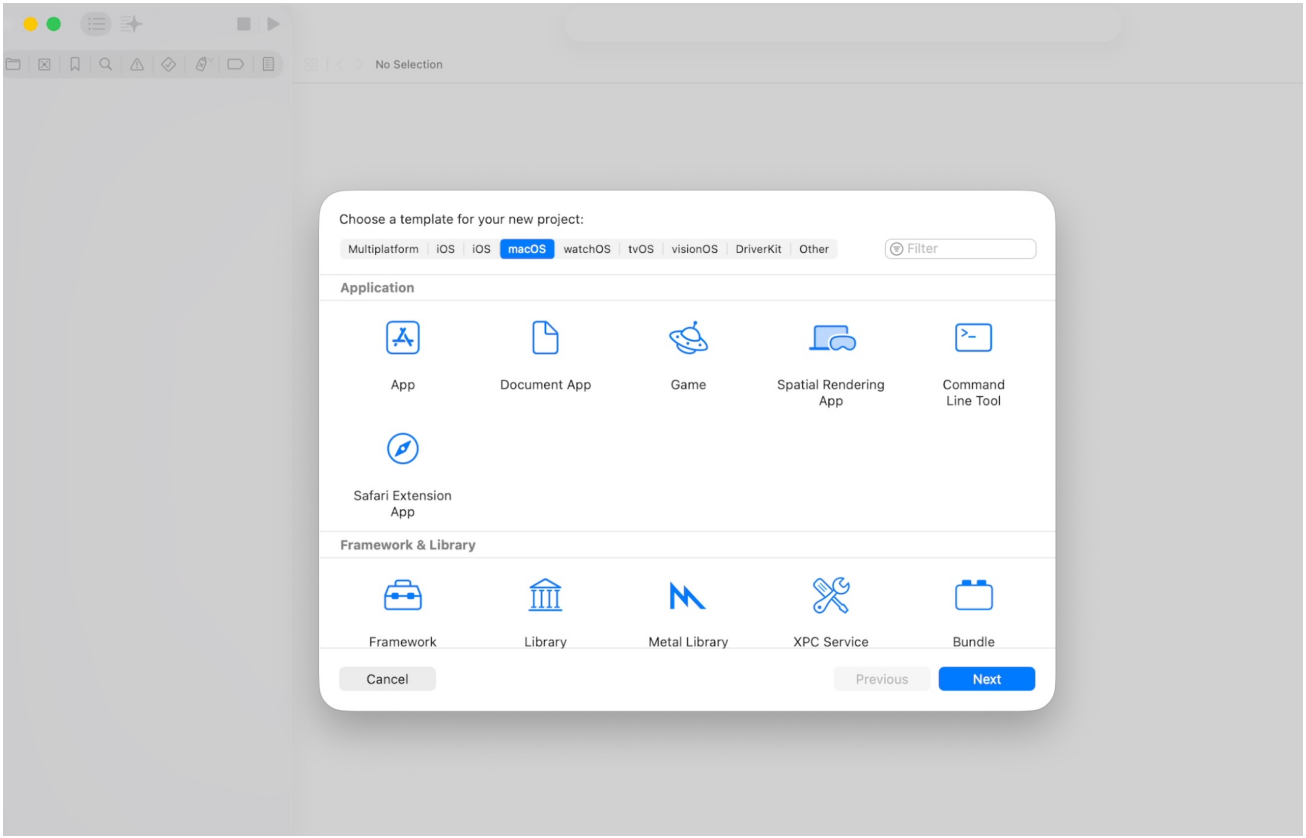
```
?  Packages git:(stage/02-add—to-workspace)  ? xed .
?  Packages git:(stage/02-add—to-workspace)  ?
```

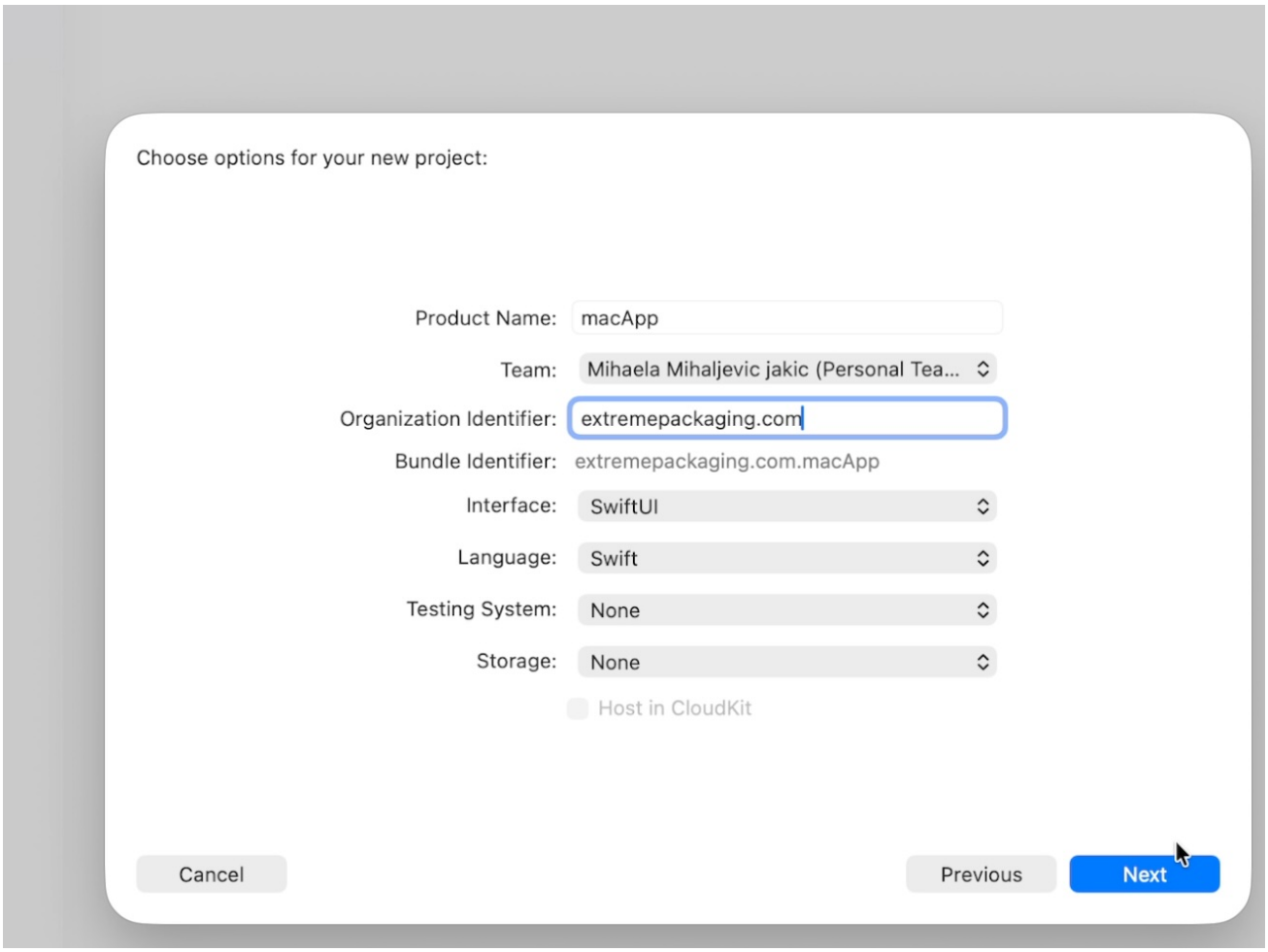
1.3 Stage 03 — Add iOS and macOS App Targets

Platform targets creation

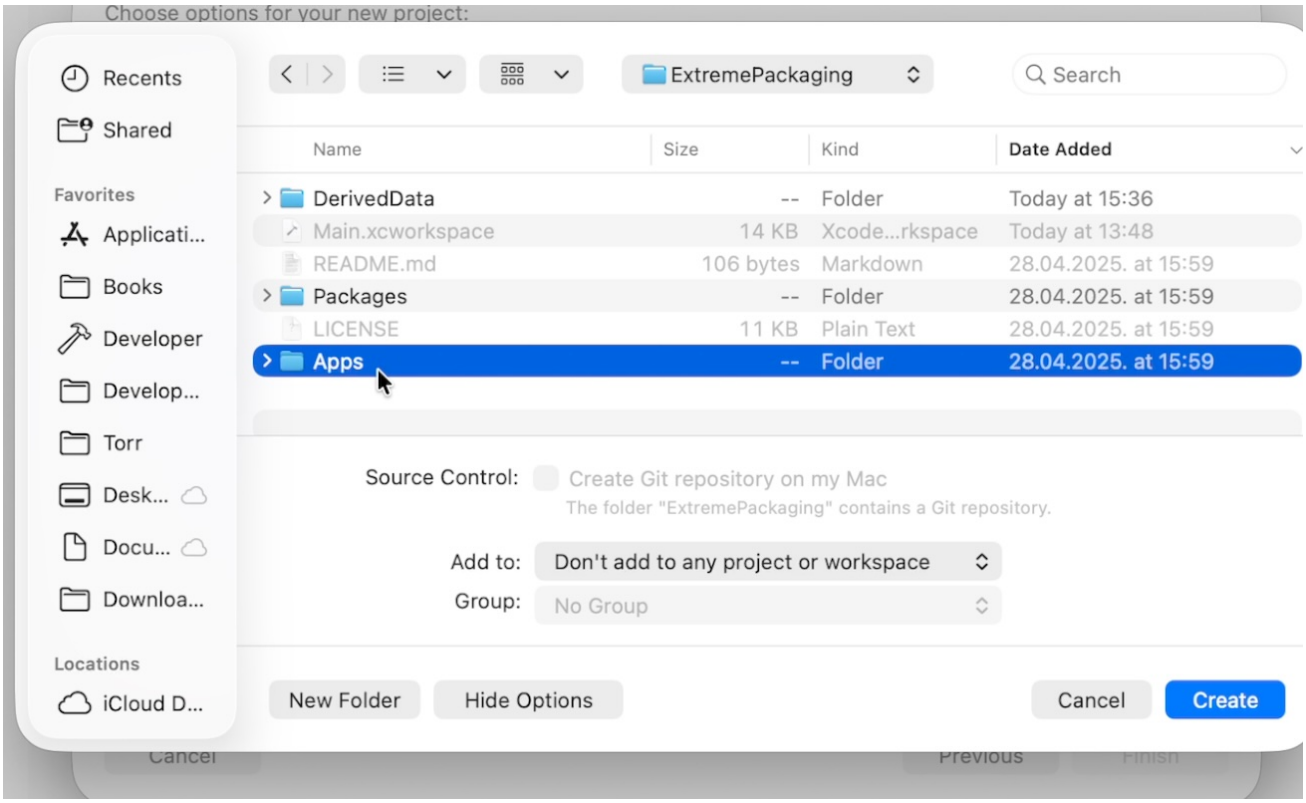
Add new iOS and a new macOS project Repeat this for each target:

Create a new project in Xcode:





Add each to Apps folder:

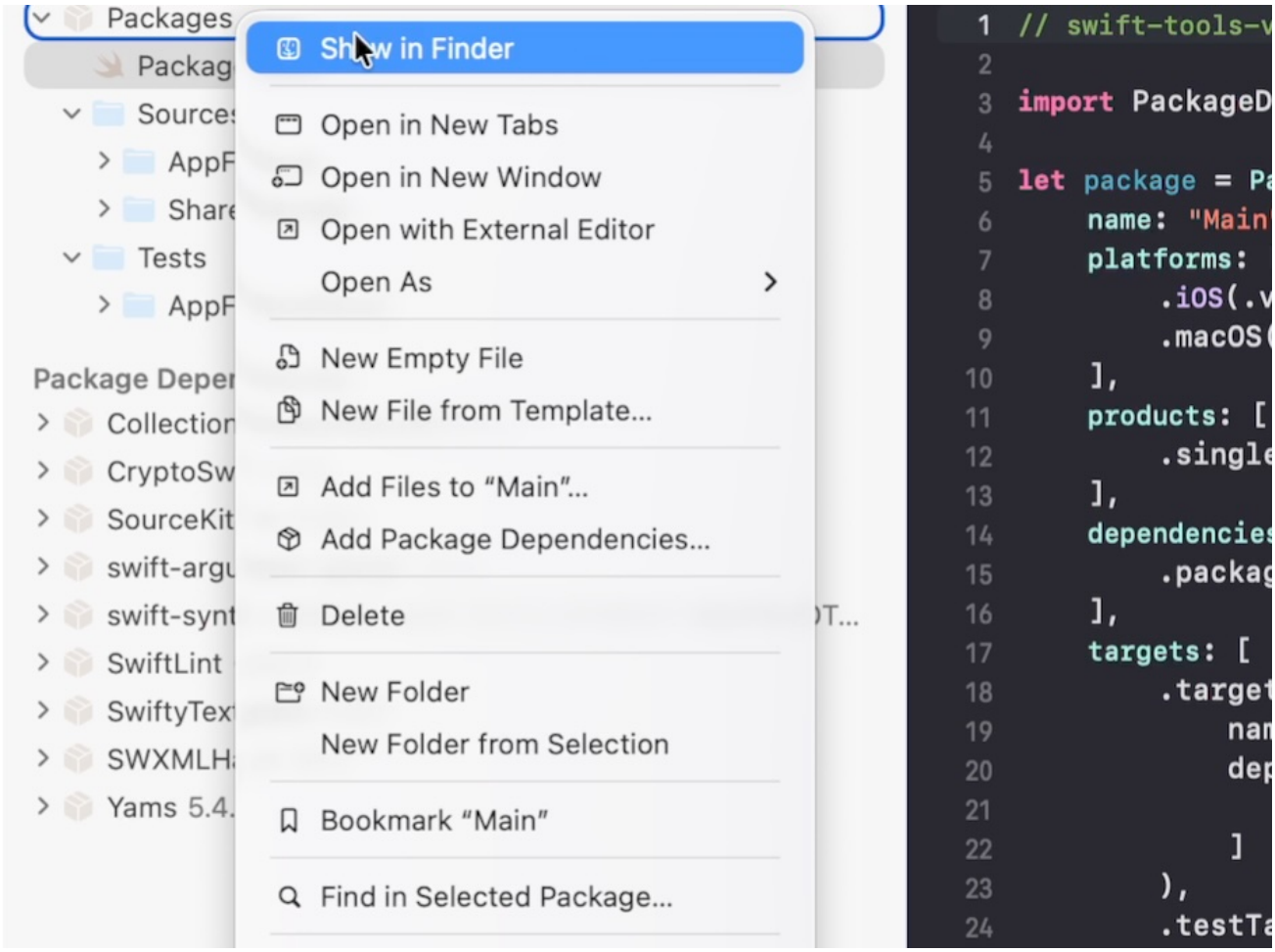


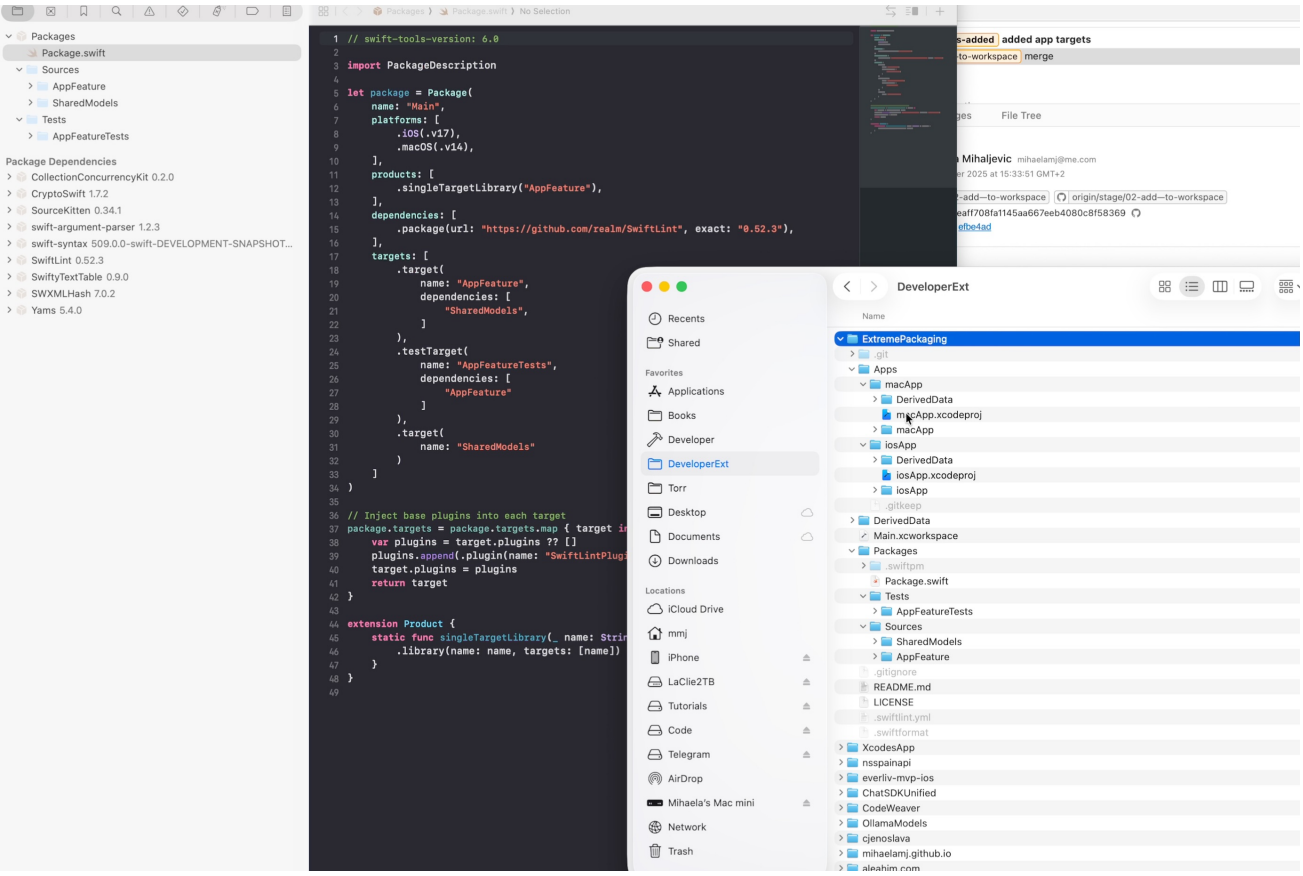
Adding them to workspace

We will be adding each app target to the workspace, below *Packages*. Open the current repository folder and drag both new projects (iOS and macOS) into the workspace sidebar.

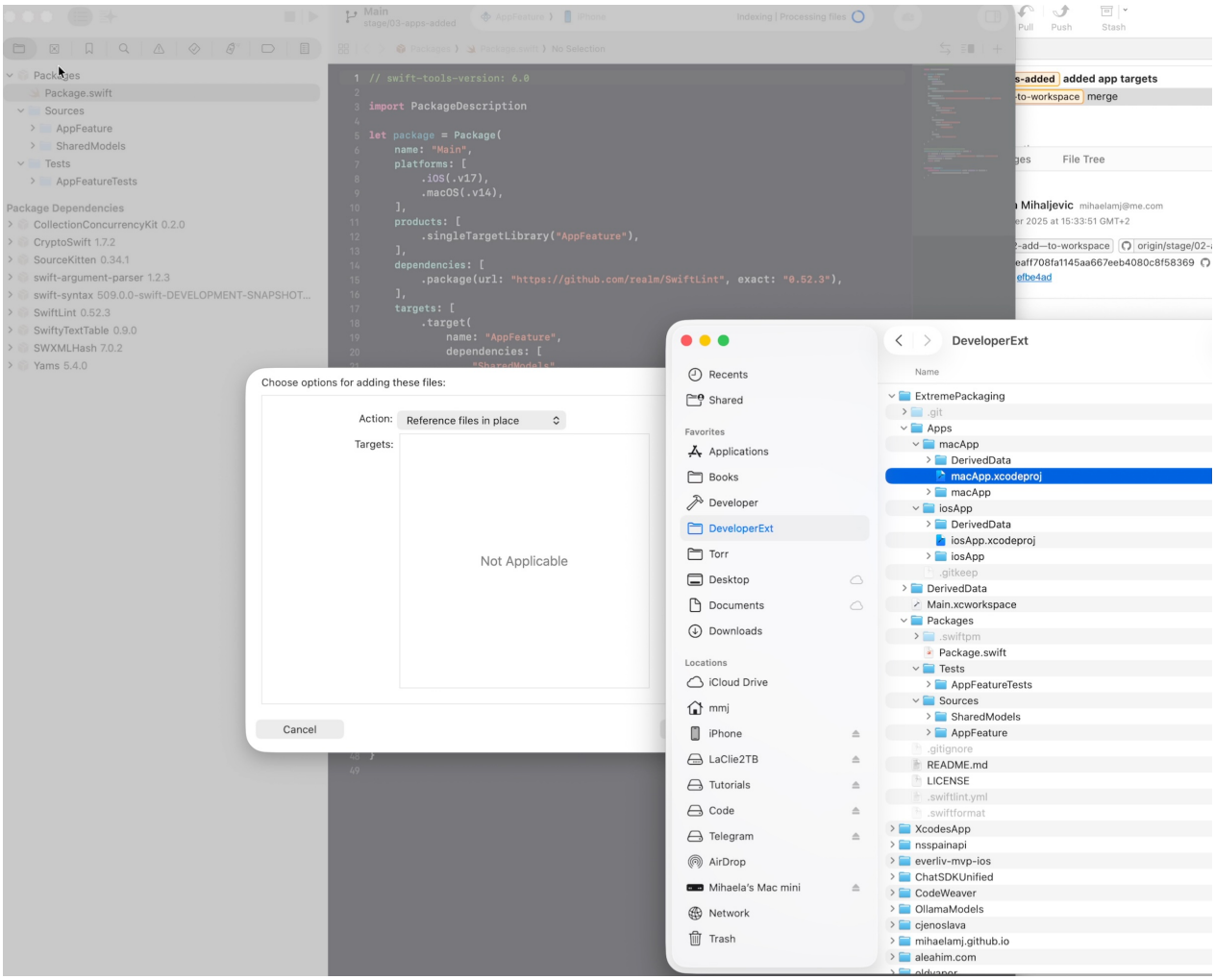
Each target remains isolated within its own folder, but both share the same logic through the *AppFeature* module. This ensures that all common code — views, models, and reducers — stays within shared packages, while each platform target keeps its own configuration files.

Open the current repo folder:

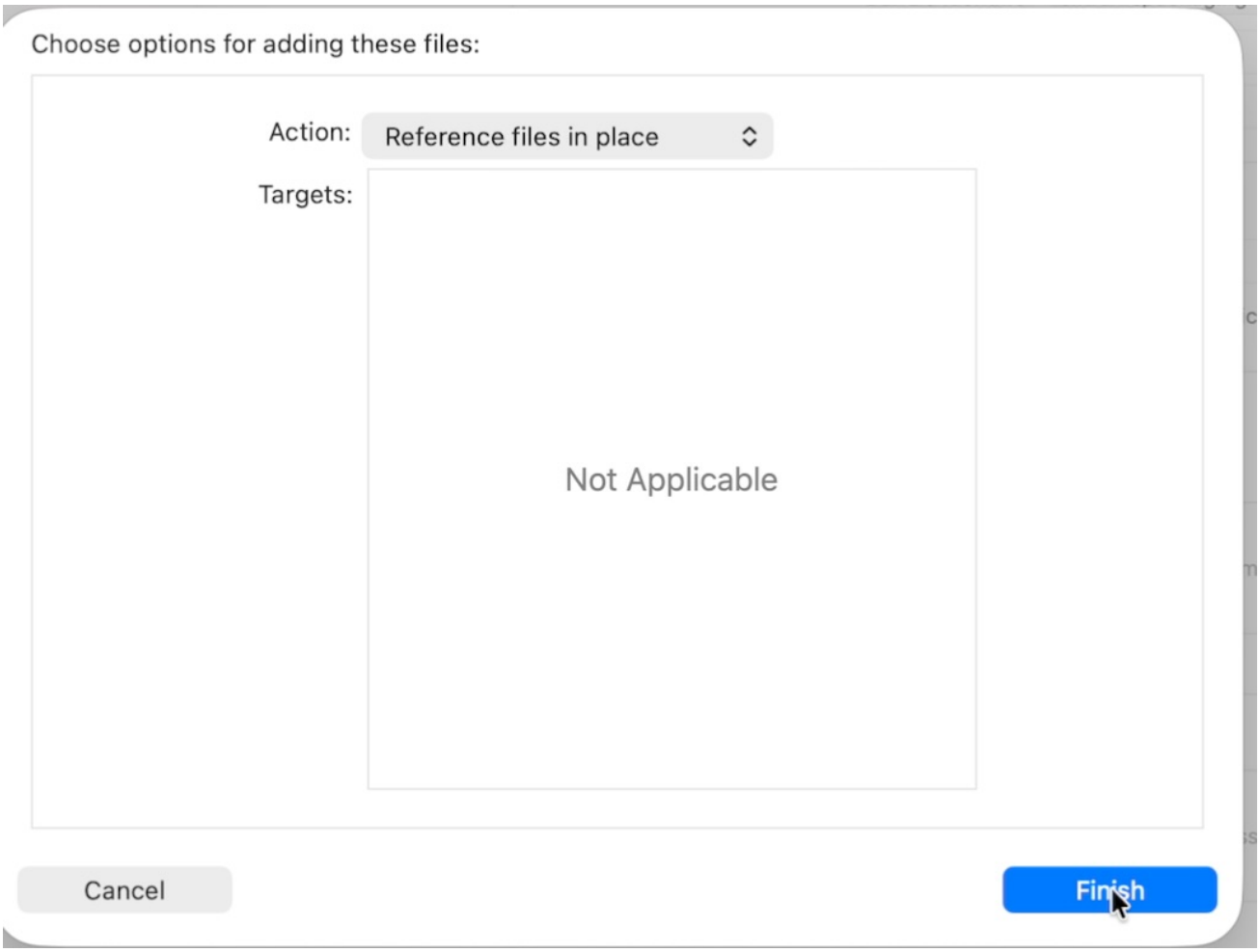




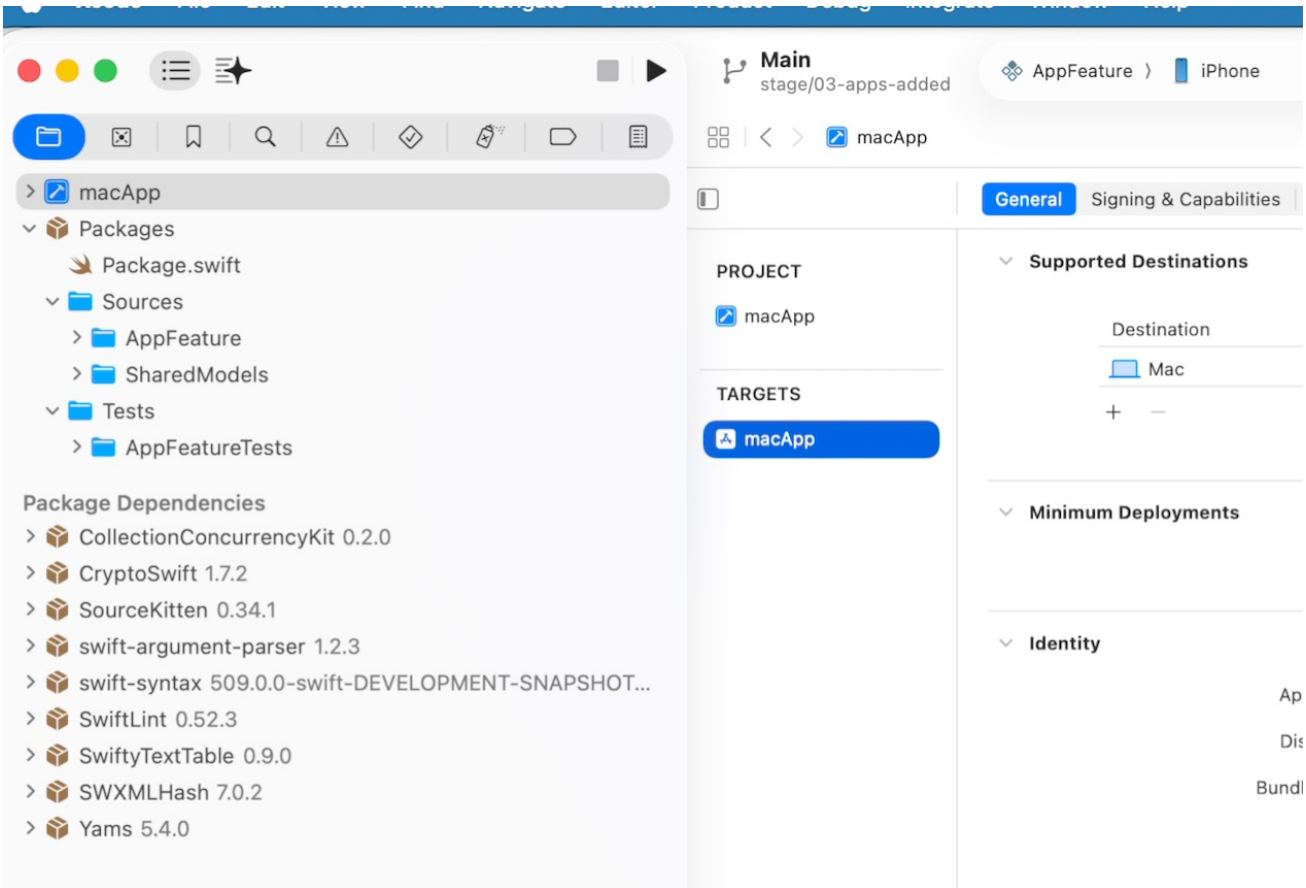
Add **macOS** target:



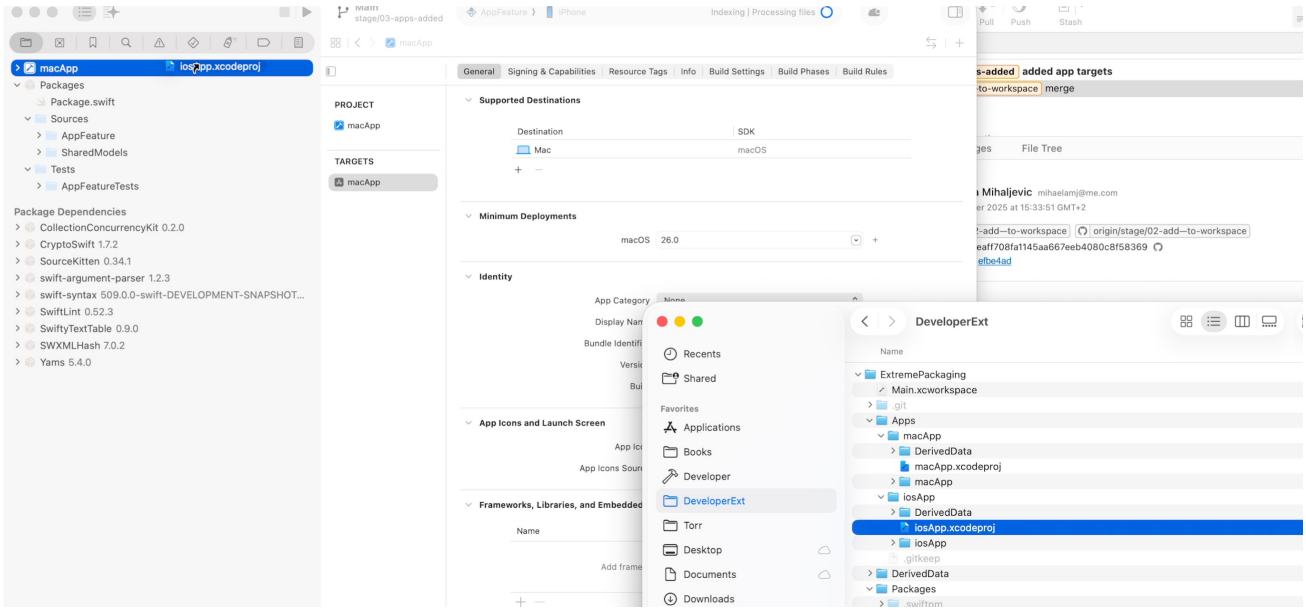
Choose “Reference files in place”



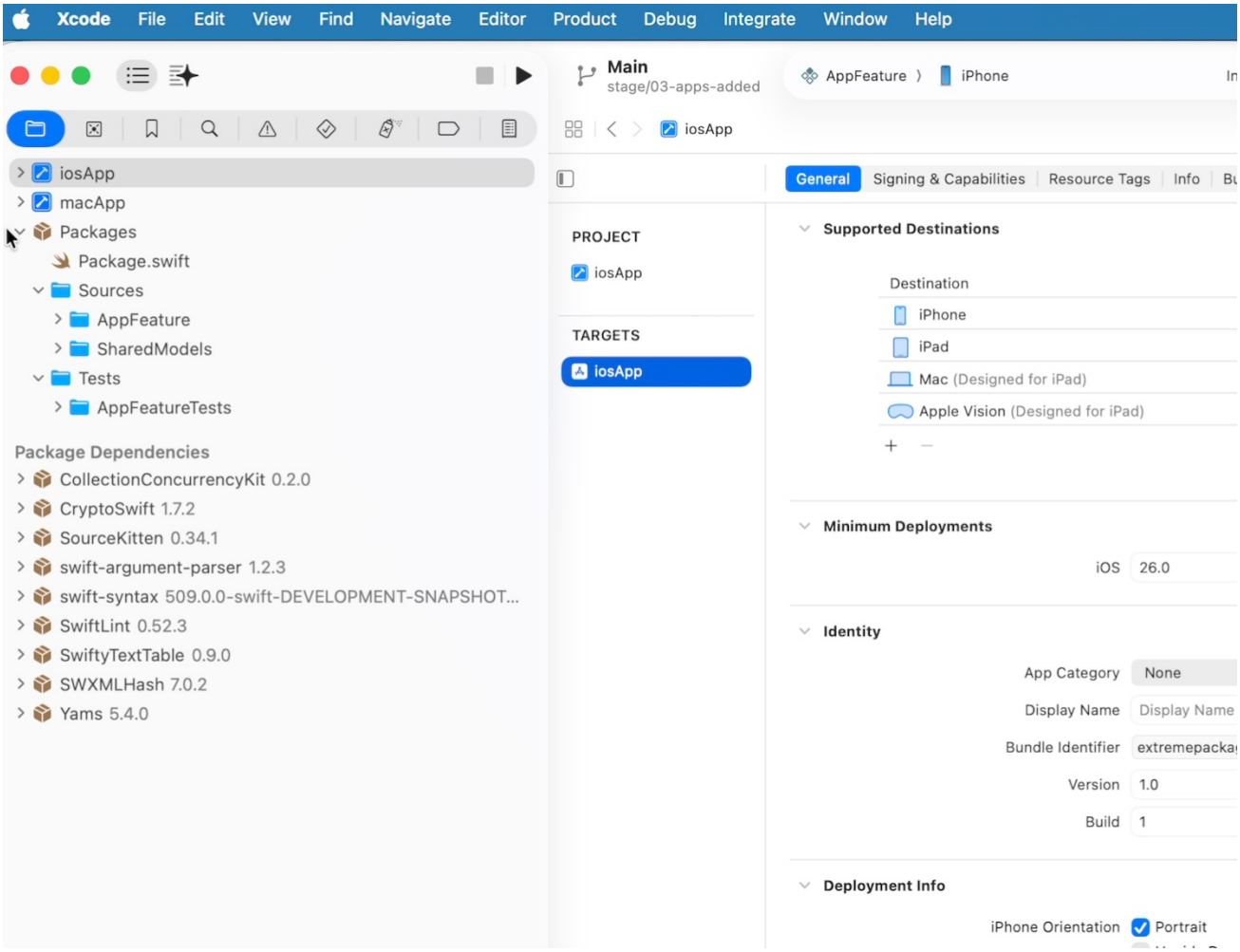
This is how it looks when added:



Add **iOS** target:



This is how it looks when added:



Explanation: Why separate targets

Keeping iOS and macOS projects distinct allows:

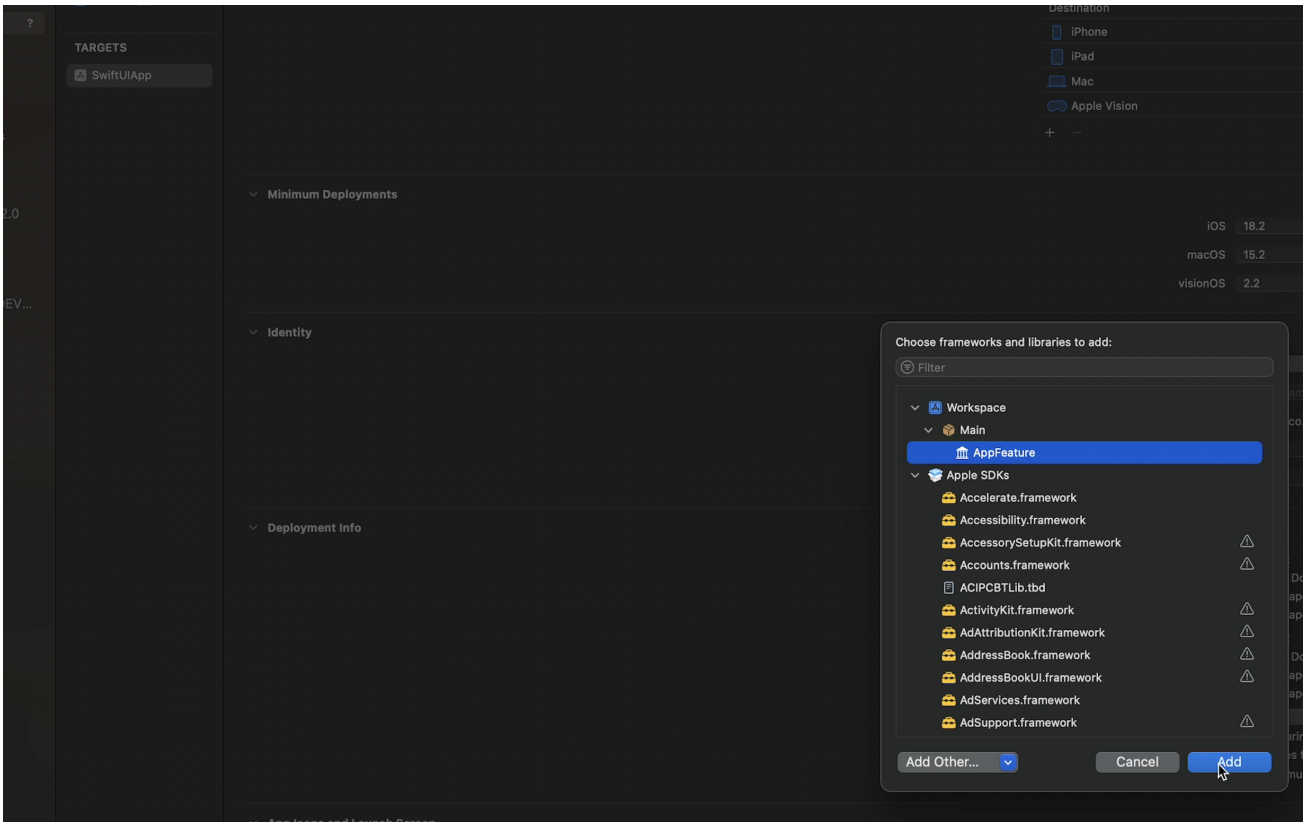
Independent platform configuration (e.g. Info.plist, app icons, signing settings)

Platform-specific features when needed (e.g. menu commands on macOS, touch gestures on iOS)
Consistent architecture and shared logic through modular Swift packages

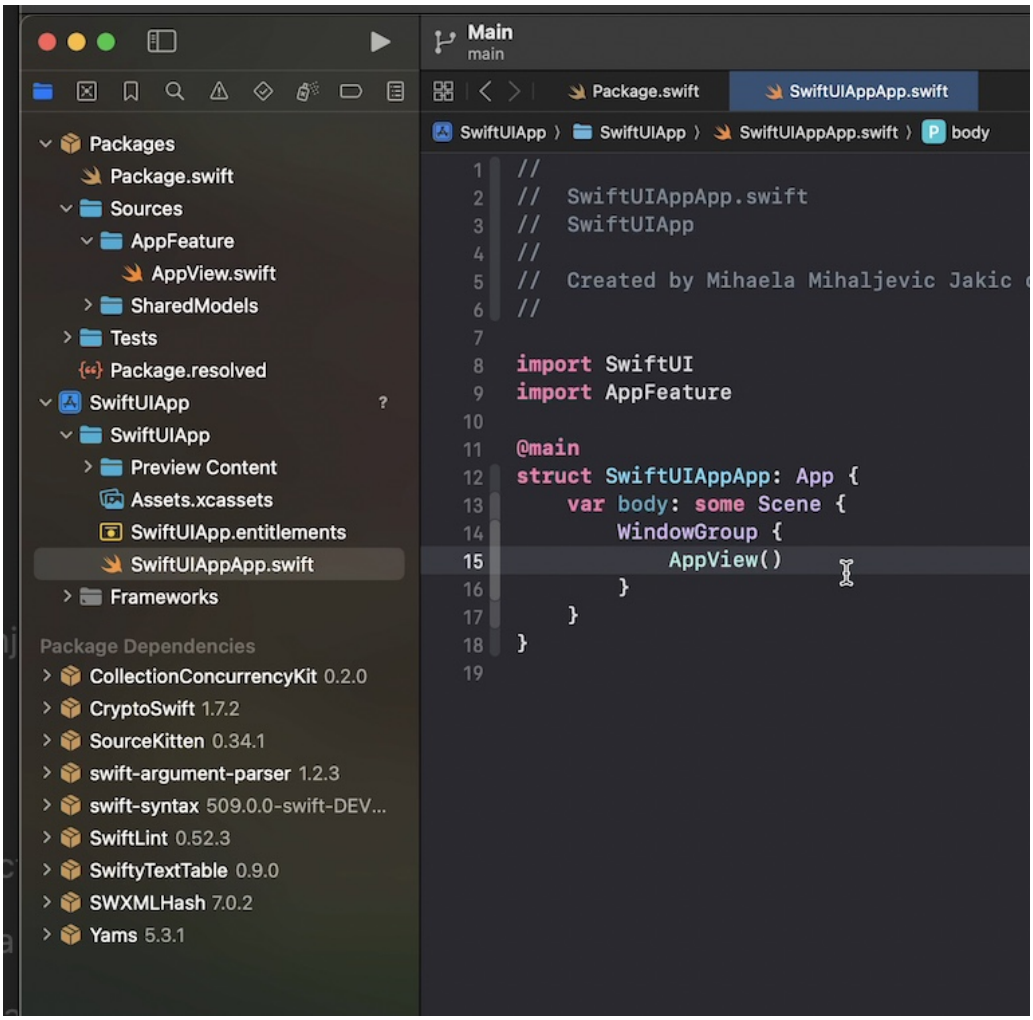
This structure aligns perfectly with the **Extreme Packaging** philosophy — shared foundation, platform-specific shells.

Configuration steps

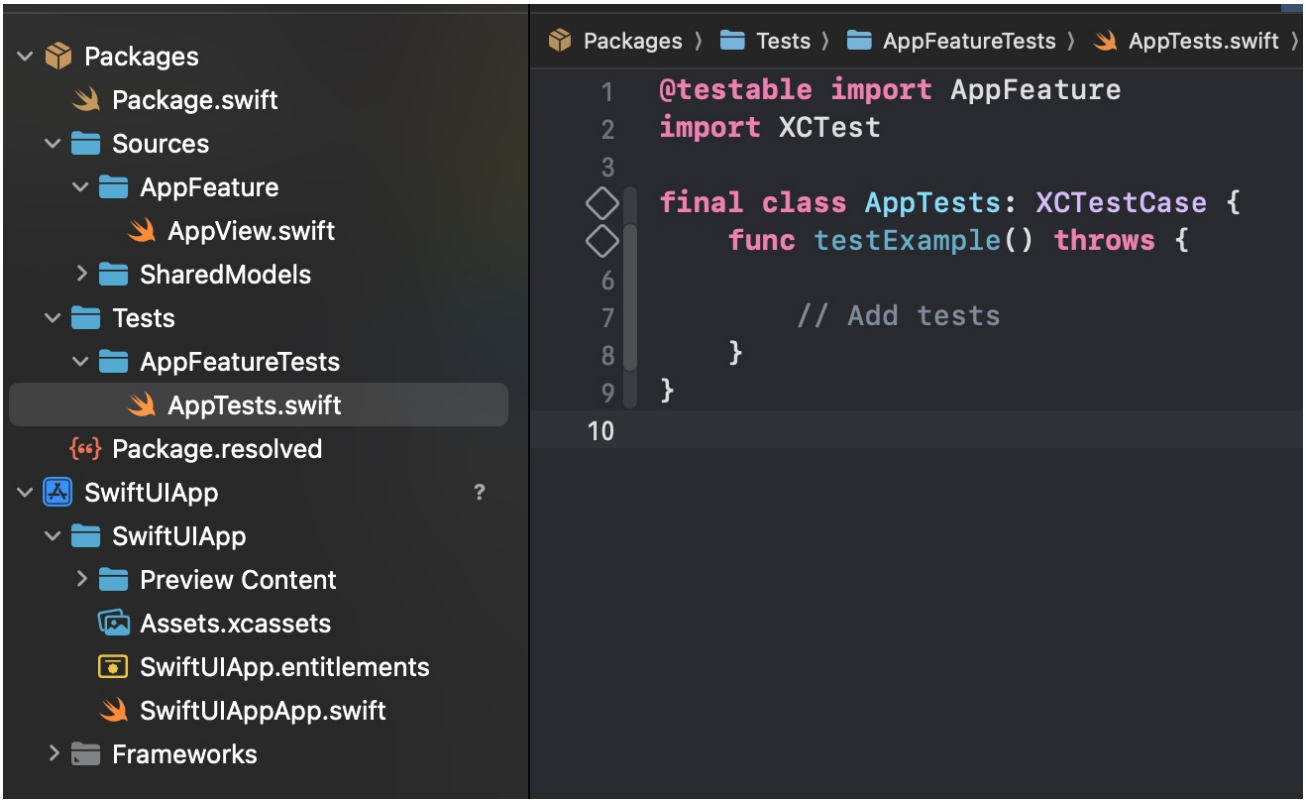
Delete automatically added `ContentView.swift`
Add `AppFeature` package to the our target:



Now our project looks like this:



And the main app starts with the fully testable `AppFeature`



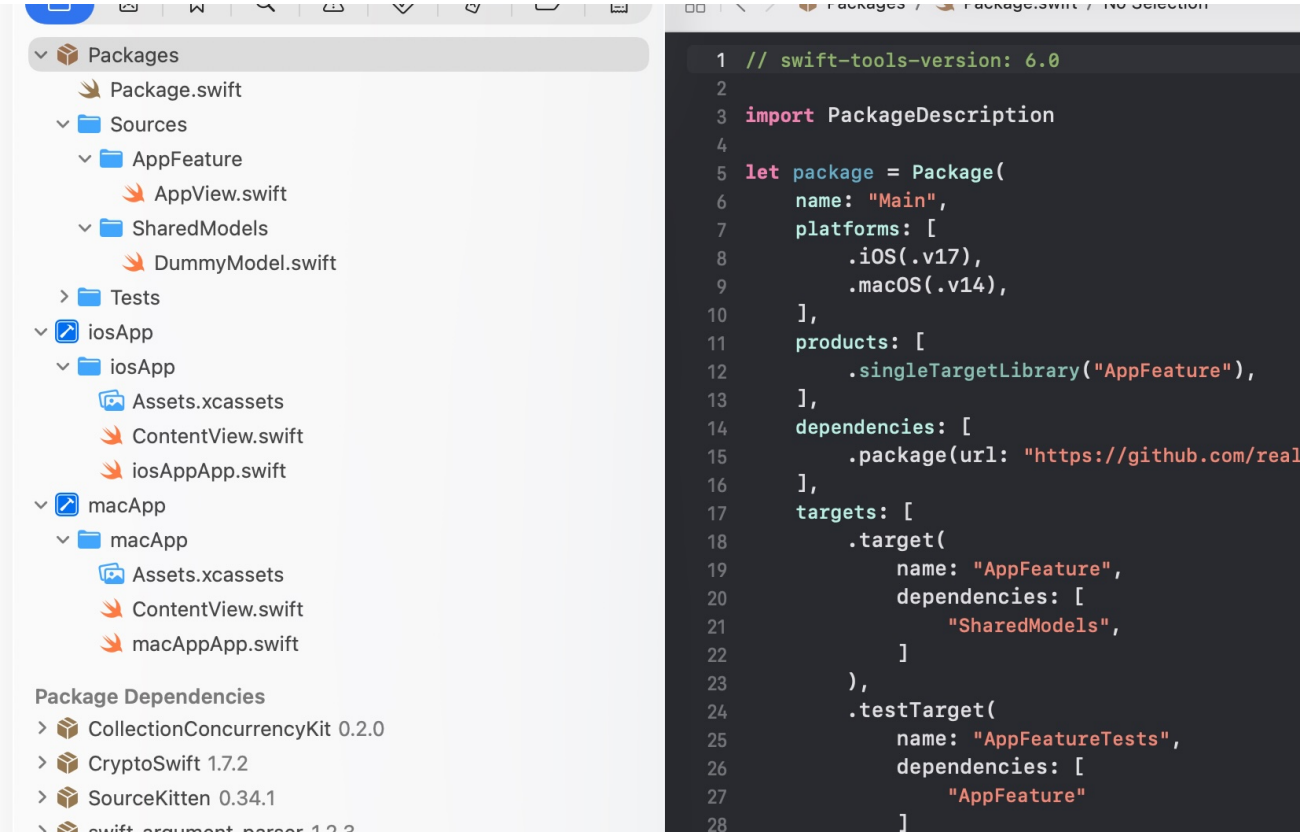
```
import SwiftUI
import AppFeature

@main
struct SwiftUIAppApp: App {
    var body: some Scene {
        WindowGroup {
            AppView()
        }
    }
}
```

Final structure

At this point, the project has evolved into a fully cross-platform modular setup.

The screenshot below illustrates the final structure after completing **Stage 03 (iOS & macOS targets added)**:



The **Packages** directory defines the shared Swift Package with `AppFeature` and `SharedModels` modules.

Both **iOS** and **macOS** apps live inside the `Apps` folder, each with its own assets, entry point, and platform-specific configuration files.

The `Package.swift` file defines a single modular product, with SwiftLint integrated as a plugin and dependencies kept cleanly separated.

This structure enables both platforms to share logic and UI built with **SwiftUI**, while still allowing native customization per platform.

Result: A clean, modular workspace where shared code resides in packages, and each platform acts only as a thin presentation shell — the essence of *Extreme Packaging*.

Stage 03-apps-added

In **stage/03-apps-added**, we create two separate app targets — one for **iOS** and one for **macOS** — both sharing the same SwiftUI core logic.

Each app lives inside the `Apps` folder and imports the `AppFeature` module, demonstrating how the same feature package can power multiple platforms with no duplicated code.

After adding the targets:

- Delete the automatically generated `ContentView.swift`
- Add the `AppFeature` package dependency to both targets
- Verify that both apps build successfully

This stage concludes with a fully functional cross-platform setup where both iOS and macOS share a unified architecture driven by modular packages.

Here's the code for **stage 03**:

```
# Clone repo if needed
git clone git@github.com:mihaelamj/ExtremePackaging.git

# Ensure you're on the branch you want
git checkout stage/03-apps-added

# Fetch the latest version
git fetch origin

# Reset your branch to the remote version
git reset --hard stage/03-apps-added

# Delete untracked files and directories
git clean -fdx
```

Summary

Stage	Description	Key Additions
01	Initial package setup	`Package.swift`, SwiftLint & SwiftFormat configs, dummy modules (`AppFeature`, `SharedModels`)
02	Workspace creation	`Main.xcworkspace`, integrated `Packages` folder for modular management
03	iOS & macOS app targets added	Separate iOS and macOS apps in `Apps/`, both using `AppFeature` for shared SwiftUI logic

Each stage corresponds to a dedicated branch in the repository, allowing you to switch between checkpoints and observe the project's evolution step by step. This structure provides a transparent history of how a modular Swift project grows from a single package into a fully multi-platform architecture.

Part 2 — Tooling

Every project includes **SwiftLint** and **SwiftFormat** for enforcing consistent code style and quality

Each stage of the repository introduces incremental improvements, from initializing packages to adding app targets and integrations

These are applied from the very first stage but can be customized at any point.

GitIgnore

```
# Xcode
#
# gitignore contributors: remember to update Global/Xcode.gitignore,
Objective-C.gitignore & Swift.gitignore

## User settings
xcuserdata/

## Obj-C/Swift specific
*.hmap

## App packaging
*.ipa
*.dSYM.zip
*.dSYM

## Playgrounds
timeline.xctimeline
playground.xcworkspace

# Swift Package Manager
.build/
.swiftpm/
Package.resolved

# Xcode workspace & projects
*.xcworkspace/xcshareddata/WorkspaceSettings.xcsettings
*.xcworkspace/xcuserdata/
DerivedData/
*.xcuserstate
*.xcscmblueprint
*.xccheckout

# macOS system files
.DS_Store

# Carthage
Carthage/Build/

# fastlane
fastlane/report.xml
```

```
fastlane/Preview.html
fastlane/screenshots/**/*.png
fastlane/test_output

# Custom folders used in ExtremePackaging
Stage/
Tasks/.build/
```

SwiftFormat Config

This is my `.swiftformat`

```
# General Options
--swiftversion 5.7

# File Options
--exclude
Stage,Tasks/.build,ThirdParty,**/SwiftGen/*,**/Sourcery/*,Frameworks/swift-compo
sable-architecture,**.generated.swift

# Format Options

--allman false
--binarygrouping none
--closingparen balanced
--commas always
--conflictmarkers reject
--decimalgrouping none
--elseposition same-line
--exponentcase lowercase
--exponentgrouping disabled
--fractiongrouping disabled
--fragment false
--header ignore
--hexgrouping none
--hexliteralcase lowercase
--ifdef no-indent
--importgrouping alphabetized
--indent 4
--indentcase false
# make sure this matches .swiftlint
--maxwidth 180
--nospaceoperators ..<,...
--octalgrouping none
--operatorfunc no-space
--selfrequired
--stripunusedargs closure-only
--trailingclosures
--wrapparguments before-first
--wrapcollections before-first
--wrapparameters before-first
```

```
# Rules
--disable hoistAwait
--disable hoistPatternLet
--disable hoistTry
--disable wrapMultilineStatementBraces
--disable extensionAccessControl
```

SwiftLint Config

```
disabled_rules: # rule identifiers to exclude from running
  - opening_brace
  - operator_whitespace
  - orphaned_doc_comment

opt_in_rules:
  - empty_count
  - force_unwrapping
  - shorthand_optional_binding
  - weak_delegate

excluded:
  - "*.generated"

custom_rules:
  # check's for Combine's .assign(to: xxx, on: self) ref-cycle
  combine_assign_to_self:
    included: ".*\\.swift"
    name: "`assign` to self"
    regex: '\.assign\(to: [^,]*, on: self\)'
    message: "For assigning on self, use assignNoRetain(to: ..., on: self)."
    severity: error
  duplicate_remove_duplicates:
    included: ".*\\.swift"
    name: "Duplicate `removeDuplicates()`"
    message: "ViewStore's publisher already does `removeDuplicates()`"
    regex: 'publisher\.[^(|{|,]*removeDuplicates\(\)'
    severity: error
  dont_scale_to_zero:
    included: ".*\\.swift"
    name: "Don't scale down to 0."
    regex: "\\.scaleEffect\\([^\\)]*(\\ 0\\ [^\\)]*\\)|0.0(\\ |\\))|\\
0(\\)|,))"
    message: "Please make sure to pass a number not equal zero, so
transformations don't throw warnings like `ignoring singular matrix`."
    severity: error
  use_data_constructor_over_string_member:
    included: ".*\\.swift"
    name: "Do not use String.data(using: .utf8)"
    regex: "\\.?data\\(using: \\.utf8\\)"
```

```
    message: "Please use Data(string.utf8) instead of String.data(using: .utf8)
because the Data constructor is non-optional and Strings are guaranteed to be
encodable as .utf8"
    severity: error
  tca_explicit_generics_reducer:
    included: ".*\\.swift"
    name: "Explicit Generics for Reducer"
    regex: 'Reduce\s+\{'
    message: "Use explicit generics in ReducerBuilder (Reduce<State, Action>)
for successful autocompletion."
    severity: error
  tca_scope_unused_closure_parameter:
    name: "TCA Scope Unused Closure Parameter"
    regex: '\.scope\(\s*state\s*:\s*\{\s*\_'
    message: "Explicitly use closure parameter when scoping store (ensures the
right state is being mutated)"
    severity: error
  tca_use_observe_viewstore_api:
    name: "TCA ViewStore observe API"
    regex: 'ViewStore\(store\.scope'
    message: "Use modern observe: api instead of store.scope"
    severity: error

trailing_comma:
    mandatory_comma: true

cyclomatic_complexity:
  ignores_case_statements: true
  warning: 20

file_length:
  warning: 1000
  error: 1000

identifier_name:
  severity: warning
  allowed_symbols: "_"
  min_length: 2
  max_length:
    warning: 90
    error: 90
  excluded:
    - iO
    - id
    - vc
    - x
    - y
    - i
    - pi
    - d

legacy_constant: error
```

```
legacy_constructor: error

line_length:
  warning: 180
  error: 180
  ignores_comments: true
  ignores_urls: true

nesting:
  type_level:
    warning: 3
    error: 3
  function_level:
    warning: 5
    error: 5

function_parameter_count:
  warning: 5

force_cast: warning
force_unwrapping: warning

type_body_length:
  - 300 # warning
  - 300 # error

large_tuple:
  - 3  # warning
  - 10 # error%
```